**Spring Cloud**

**Table of Contents**

[**1. Features**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_features)

[**I. Cloud Native Applications**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_cloud_native_applications)

[**2. Spring Cloud Context: Application Context Services**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_cloud_context_application_context_services)

[**2.1. The Bootstrap Application Context**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_the_bootstrap_application_context)

[**2.2. Application Context Hierarchies**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_application_context_hierarchies)

[**2.3. Changing the Location of Bootstrap Properties**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#customizing-bootstrap-properties)

[**2.4. Overriding the Values of Remote Properties**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#overriding-bootstrap-properties)

[**2.5. Customizing the Bootstrap Configuration**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_customizing_the_bootstrap_configuration)

[**2.6. Customizing the Bootstrap Property Sources**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#customizing-bootstrap-property-sources)

[**2.7. Logging Configuration**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_logging_configuration)

[**2.8. Environment Changes**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_environment_changes)

[**2.9. Refresh Scope**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#refresh-scope)

[**2.10. Encryption and Decryption**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_encryption_and_decryption)

[**2.11. Endpoints**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_endpoints)

[**3. Spring Cloud Commons: Common Abstractions**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_cloud_commons_common_abstractions)

[**3.1. @EnableDiscoveryClient**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#__enablediscoveryclient)

[3.1.1. Health Indicator](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_health_indicator)

[**3.2. ServiceRegistry**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_serviceregistry)

[3.2.1. ServiceRegistry Auto-Registration](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_serviceregistry_auto_registration)

[3.2.2. Service Registry Actuator Endpoint](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_service_registry_actuator_endpoint)

[**3.3. Spring RestTemplate as a Load Balancer Client**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_resttemplate_as_a_load_balancer_client)

[**3.4. Spring WebClient as a Load Balancer Client**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_webclient_as_a_load_balancer_client)

[3.4.1. Retrying Failed Requests](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_retrying_failed_requests)

[**3.5. Multiple RestTemplate objects**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_multiple_resttemplate_objects)

[**3.6. Spring WebFlux WebClient as a Load Balancer Client**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#loadbalanced-webclient)

[**3.7. Ignore Network Interfaces**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#ignore-network-interfaces)

[**3.8. HTTP Client Factories**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#http-clients)

[**3.9. Enabled Features**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#enabled-features)

[3.9.1. Feature types](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_feature_types)

[3.9.2. Declaring features](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_declaring_features)

[**II. Spring Cloud Config**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_cloud_config)

[**4. Quick Start**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_quick_start)

[**4.1. Client Side Usage**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_client_side_usage)

[**5. Spring Cloud Config Server**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_cloud_config_server)

[**5.1. Environment Repository**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_environment_repository)

[5.1.1. Git Backend](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_git_backend)

[Skipping SSL Certificate Validation](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_skipping_ssl_certificate_validation)

[Setting HTTP Connection Timeout](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_setting_http_connection_timeout)

[Placeholders in Git URI](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_placeholders_in_git_uri)

[Pattern Matching and Multiple Repositories](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_pattern_matching_and_multiple_repositories)

[Authentication](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_authentication)

[Authentication with AWS CodeCommit](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_authentication_with_aws_codecommit)

[Git SSH configuration using properties](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_git_ssh_configuration_using_properties)

[Placeholders in Git Search Paths](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_placeholders_in_git_search_paths)

[Force pull in Git Repositories](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_force_pull_in_git_repositories)

[Deleting untracked branches in Git Repositories](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_deleting_untracked_branches_in_git_repositories)

[5.1.2. Version Control Backend Filesystem Use](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_version_control_backend_filesystem_use)

[5.1.3. File System Backend](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_file_system_backend)

[5.1.4. Vault Backend](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault-backend)

[Multiple Properties Sources](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_multiple_properties_sources)

[5.1.5. Accessing Backends Through a Proxy](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_accessing_backends_through_a_proxy)

[5.1.6. Sharing Configuration With All Applications](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_sharing_configuration_with_all_applications)

[File Based Repositories](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-config-server-file-based-repositories)

[Vault Server](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-config-server-vault-server)

[5.1.7. JDBC Backend](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_jdbc_backend)

[5.1.8. Composite Environment Repositories](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#composite-environment-repositories)

[Custom Composite Environment Repositories](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_custom_composite_environment_repositories)

[5.1.9. Property Overrides](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_property_overrides)

[**5.2. Health Indicator**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_health_indicator_2)

[**5.3. Security**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_security)

[**5.4. Encryption and Decryption**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_encryption_and_decryption_2)

[**5.5. Key Management**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_key_management)

[**5.6. Creating a Key Store for Testing**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_creating_a_key_store_for_testing)

[**5.7. Using Multiple Keys and Key Rotation**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_using_multiple_keys_and_key_rotation)

[**5.8. Serving Encrypted Properties**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_serving_encrypted_properties)

[**6. Serving Alternative Formats**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_serving_alternative_formats)

[**7. Serving Plain Text**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_serving_plain_text)

[**8. Embedding the Config Server**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_embedding_the_config_server)

[**9. Push Notifications and Spring Cloud Bus**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_push_notifications_and_spring_cloud_bus)

[**10. Spring Cloud Config Client**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_cloud_config_client)

[**10.1. Config First Bootstrap**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#config-first-bootstrap)

[**10.2. Discovery First Bootstrap**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#discovery-first-bootstrap)

[**10.3. Config Client Fail Fast**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#config-client-fail-fast)

[**10.4. Config Client Retry**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#config-client-retry)

[**10.5. Locating Remote Configuration Resources**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_locating_remote_configuration_resources)

[**10.6. Specifying Multiple Urls for the Config Server**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_specifying_multiple_urls_for_the_config_server)

[**10.7. Configuring Read Timeouts**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_configuring_read_timeouts)

[**10.8. Security**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_security_2)

[10.8.1. Health Indicator](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_health_indicator_3)

[10.8.2. Providing A Custom RestTemplate](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#custom-rest-template)

[10.8.3. Vault](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_vault)

[**10.9. Nested Keys In Vault**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_nested_keys_in_vault)

[**III. Spring Cloud Netflix**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_cloud_netflix)

[**11. Service Discovery: Eureka Clients**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_service_discovery_eureka_clients)

[**11.1. How to Include Eureka Client**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#netflix-eureka-client-starter)

[**11.2. Registering with Eureka**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_registering_with_eureka)

[**11.3. Authenticating with the Eureka Server**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_authenticating_with_the_eureka_server)

[**11.4. Status Page and Health Indicator**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_status_page_and_health_indicator)

[**11.5. Registering a Secure Application**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_registering_a_secure_application)

[**11.6. Eureka’s Health Checks**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_eureka_s_health_checks)

[**11.7. Eureka Metadata for Instances and Clients**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_eureka_metadata_for_instances_and_clients)

[11.7.1. Using Eureka on Cloud Foundry](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_using_eureka_on_cloud_foundry)

[11.7.2. Using Eureka on AWS](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_using_eureka_on_aws)

[11.7.3. Changing the Eureka Instance ID](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_changing_the_eureka_instance_id)

[**11.8. Using the EurekaClient**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_using_the_eurekaclient)

[11.8.1. EurekaClient without Jersey](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_eurekaclient_without_jersey)

[**11.9. Alternatives to the Native Netflix EurekaClient**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_alternatives_to_the_native_netflix_eurekaclient)

[**11.10. Why Is It so Slow to Register a Service?**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_why_is_it_so_slow_to_register_a_service)

[**11.11. Zones**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_zones)

[**12. Service Discovery: Eureka Server**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-eureka-server)

[**12.1. How to Include Eureka Server**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#netflix-eureka-server-starter)

[**12.2. How to Run a Eureka Server**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-running-eureka-server)

[**12.3. High Availability, Zones and Regions**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-eureka-server-zones-and-regions)

[**12.4. Standalone Mode**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-eureka-server-standalone-mode)

[**12.5. Peer Awareness**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-eureka-server-peer-awareness)

[**12.6. When to Prefer IP Address**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-eureka-server-prefer-ip-address)

[**12.7. Securing The Eureka Server**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_securing_the_eureka_server)

[**13. Circuit Breaker: Hystrix Clients**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_circuit_breaker_hystrix_clients)

[**13.1. How to Include Hystrix**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_how_to_include_hystrix)

[**13.2. Propagating the Security Context or Using Spring Scopes**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#netflix-hystrix-starter)

[**13.3. Health Indicator**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_health_indicator_4)

[**13.4. Hystrix Metrics Stream**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_hystrix_metrics_stream)

[**14. Circuit Breaker: Hystrix Dashboard**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_circuit_breaker_hystrix_dashboard)

[**15. Hystrix Timeouts And Ribbon Clients**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_hystrix_timeouts_and_ribbon_clients)

[**15.1. How to Include the Hystrix Dashboard**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#netflix-hystrix-dashboard-starter)

[**15.2. Turbine**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_turbine)

[15.2.1. Clusters Endpoint](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_clusters_endpoint)

[**15.3. Turbine Stream**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_turbine_stream)

[**16. Client Side Load Balancer: Ribbon**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-ribbon)

[**16.1. How to Include Ribbon**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#netflix-ribbon-starter)

[**16.2. Customizing the Ribbon Client**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_customizing_the_ribbon_client)

[**16.3. Customizing the Default for All Ribbon Clients**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_customizing_the_default_for_all_ribbon_clients)

[**16.4. Customizing the Ribbon Client by Setting Properties**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_customizing_the_ribbon_client_by_setting_properties)

[**16.5. Using Ribbon with Eureka**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_using_ribbon_with_eureka)

[**16.6. Example: How to Use Ribbon Without Eureka**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-ribbon-without-eureka)

[**16.7. Example: Disable Eureka Use in Ribbon**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_example_disable_eureka_use_in_ribbon)

[**16.8. Using the Ribbon API Directly**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_using_the_ribbon_api_directly)

[**16.9. Caching of Ribbon Configuration**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#ribbon-child-context-eager-load)

[**16.10. How to Configure Hystrix Thread Pools**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#how-to-configure-hystrix-thread-pools)

[**16.11. How to Provide a Key to Ribbon’s IRule**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#how-to-provdie-a-key-to-ribbon)

[**17. External Configuration: Archaius**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_external_configuration_archaius)

[**18. Router and Filter: Zuul**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_router_and_filter_zuul)

[**18.1. How to Include Zuul**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#netflix-zuul-starter)

[**18.2. Embedded Zuul Reverse Proxy**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#netflix-zuul-reverse-proxy)

[**18.3. Zuul Http Client**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_zuul_http_client)

[**18.4. Cookies and Sensitive Headers**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_cookies_and_sensitive_headers)

[**18.5. Ignored Headers**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_ignored_headers)

[**18.6. Management Endpoints**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_management_endpoints)

[18.6.1. Routes Endpoint](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_routes_endpoint)

[18.6.2. Filters Endpoint](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_filters_endpoint)

[**18.7. Strangulation Patterns and Local Forwards**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_strangulation_patterns_and_local_forwards)

[**18.8. Uploading Files through Zuul**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_uploading_files_through_zuul)

[**18.9. Query String Encoding**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_query_string_encoding)

[**18.10. Plain Embedded Zuul**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_plain_embedded_zuul)

[**18.11. Disable Zuul Filters**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_disable_zuul_filters)

[**18.12. Providing Hystrix Fallbacks For Routes**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#hystrix-fallbacks-for-routes)

[**18.13. Zuul Timeouts**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_zuul_timeouts)

[**18.14. Rewriting the Location header**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#zuul-redirect-location-rewrite)

[**18.15. Metrics**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_metrics)

[**18.16. Zuul Developer Guide**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#zuul-developer-guide)

[18.16.1. The Zuul Servlet](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_the_zuul_servlet)

[18.16.2. Zuul RequestContext](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_zuul_requestcontext)

[18.16.3. @EnableZuulProxy vs. @EnableZuulServer](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#__literal_enablezuulproxy_literal_vs_literal_enablezuulserver_literal)

[18.16.4. @EnableZuulServer Filters](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#zuul-developer-guide-enable-filters)

[18.16.5. @EnableZuulProxy Filters](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#__literal_enablezuulproxy_literal_filters)

[18.16.6. Custom Zuul Filter Examples](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_custom_zuul_filter_examples)

[How to Write a Pre Filter](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#zuul-developer-guide-sample-pre-filter)

[How to Write a Route Filter](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#zuul-developer-guide-sample-route-filter)

[How to Write a Post Filter](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#zuul-developer-guide-sample-post-filter)

[18.16.7. How Zuul Errors Work](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_how_zuul_errors_work)

[18.16.8. Zuul Eager Application Context Loading](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_zuul_eager_application_context_loading)

[**19. Polyglot support with Sidecar**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_polyglot_support_with_sidecar)

[**20. Retrying Failed Requests**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#retrying-failed-requests)

[**20.1. BackOff Policies**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_backoff_policies)

[**20.2. Configuration**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_configuration)

[20.2.1. Zuul](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_zuul)

[**21. HTTP Clients**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_http_clients)

[**IV. Spring Cloud OpenFeign**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_cloud_openfeign)

[**22. Declarative REST Client: Feign**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-feign)

[**22.1. How to Include Feign**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#netflix-feign-starter)

[**22.2. Overriding Feign Defaults**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-feign-overriding-defaults)

[**22.3. Creating Feign Clients Manually**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_creating_feign_clients_manually)

[**22.4. Feign Hystrix Support**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-feign-hystrix)

[**22.5. Feign Hystrix Fallbacks**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-feign-hystrix-fallback)

[**22.6. Feign and @Primary**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_feign_and_literal_primary_literal)

[**22.7. Feign Inheritance Support**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-feign-inheritance)

[**22.8. Feign request/response compression**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_feign_request_response_compression)

[**22.9. Feign logging**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_feign_logging)

[**V. Spring Cloud Stream**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_cloud_stream)

[**23. Quick Start**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_quick_start_2)

[**23.1. Creating a Sample Application by Using Spring Initializr**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-preface-creating-sample-application)

[**23.2. Importing the Project into Your IDE**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-preface-importing-project)

[**23.3. Adding a Message Handler, Building, and Running**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-preface-adding-message-handler)

[**24. What’s New in 2.0?**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_what_s_new_in_2_0)

[**24.1. New Features and Components**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-preface-new-features)

[**24.2. Notable Enhancements**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-preface-notable-enhancements)

[24.2.1. Both Actuator and Web Dependencies Are Now Optional](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-preface-actuator-web-dependencies)

[24.2.2. Content-type Negotiation Improvements](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-preface-content-type-negotiation-improvements)

[**24.3. Notable Deprecations**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-preface-notable-deprecations)

[24.3.1. Java Serialization (Java Native and Kryo)](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-preface-deprecation-java-serialization)

[24.3.2. Deprecated Classes and Methods](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-preface-deprecation-classes-methods)

[**25. Introducing Spring Cloud Stream**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-introducing)

[**26. Main Concepts**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_main_concepts)

[**26.1. Application Model**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-application-model)

[26.1.1. Fat JAR](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_fat_jar)

[**26.2. The Binder Abstraction**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-binder-abstraction)

[**26.3. Persistent Publish-Subscribe Support**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-persistent-publish-subscribe-support)

[**26.4. Consumer Groups**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#consumer-groups)

[**26.5. Consumer Types**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#consumer-types)

[26.5.1. Durability](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#durability)

[**26.6. Partitioning Support**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#partitioning)

[**27. Programming Model**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_programming_model)

[**27.1. Destination Binders**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_destination_binders)

[**27.2. Destination Bindings**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_destination_bindings)

[**27.3. Producing and Consuming Messages**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-producing-consuming-messages)

[27.3.1. Spring Integration Support](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_integration_support)

[27.3.2. Using @StreamListener Annotation](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_using_streamlistener_annotation)

[27.3.3. Using @StreamListener for Content-based routing](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_using_streamlistener_for_content_based_routing)

[27.3.4. Using Polled Consumers](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-streams-overview-using-polled-consumers)

[**27.4. Error Handling**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-error-handling)

[27.4.1. Application Error Handling](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_application_error_handling)

[27.4.2. System Error Handling](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_system_error_handling)

[Drop Failed Messages](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_drop_failed_messages)

[DLQ - Dead Letter Queue](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_dlq_dead_letter_queue)

[Re-queue Failed Messages](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_re_queue_failed_messages)

[27.4.3. Retry Template](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_retry_template)

[**27.5. Reactive Programming Support**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-reactive-programming-support)

[27.5.1. Reactor-based Handlers](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_reactor_based_handlers)

[27.5.2. Reactive Sources](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_reactive_sources)

[**28. Binders**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-binders)

[**28.1. Producers and Consumers**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_producers_and_consumers)

[**28.2. Binder SPI**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-binder-api)

[**28.3. Binder Detection**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_binder_detection)

[28.3.1. Classpath Detection](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_classpath_detection)

[**28.4. Multiple Binders on the Classpath**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#multiple-binders)

[**28.5. Connecting to Multiple Systems**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#multiple-systems)

[**28.6. Binding visualization and control**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_binding_visualization_and_control)

[**28.7. Binder Configuration Properties**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_binder_configuration_properties)

[**29. Configuration Options**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_configuration_options)

[**29.1. Binding Service Properties**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_binding_service_properties)

[**29.2. Binding Properties**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#binding-properties)

[29.2.1. Common Binding Properties](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_common_binding_properties)

[29.2.2. Consumer Properties](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_consumer_properties)

[29.2.3. Producer Properties](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_producer_properties)

[**29.3. Using Dynamically Bound Destinations**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#dynamicdestination)

[**30. Content Type Negotiation**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#content-type-management)

[**30.1. Mechanics**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_mechanics)

[30.1.1. Content Type versus Argument Type](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_content_type_versus_argument_type)

[30.1.2. Message Converters](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_message_converters)

[**30.2. Provided MessageConverters**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_provided_messageconverters)

[**30.3. User-defined Message Converters**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-user-defined-message-converters)

[**31. Schema Evolution Support**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#schema-evolution)

[**31.1. Schema Registry Client**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_schema_registry_client)

[31.1.1. Schema Registry Client Properties](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_schema_registry_client_properties)

[**31.2. Avro Schema Registry Client Message Converters**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_avro_schema_registry_client_message_converters)

[31.2.1. Avro Schema Registry Message Converter Properties](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_avro_schema_registry_message_converter_properties)

[**31.3. Apache Avro Message Converters**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_apache_avro_message_converters)

[**31.4. Converters with Schema Support**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_converters_with_schema_support)

[**31.5. Schema Registry Server**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_schema_registry_server)

[31.5.1. Schema Registry Server API](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_schema_registry_server_api)

[Registering a New Schema](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-registering-new-schema)

[Retrieving an Existing Schema by Subject, Format, and Version](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-retrieve-schema-subject-format-version)

[Retrieving an Existing Schema by Subject and Format](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-retrieve-schema-subject-format)

[Retrieving an Existing Schema by ID](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-retrieve-schema-id)

[Deleting a Schema by Subject, Format, and Version](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-deleting-schema-subject-format-version)

[Deleting a Schema by ID](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-deleting-schema-id)

[Deleting a Schema by Subject](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-deleting-schema-subject)

[31.5.2. Using Confluent’s Schema Registry](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_using_confluent_s_schema_registry)

[**31.6. Schema Registration and Resolution**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_schema_registration_and_resolution)

[31.6.1. Schema Registration Process (Serialization)](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-schema-registration-process)

[31.6.2. Schema Resolution Process (Deserialization)](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-schema-resolution-process)

[**32. Inter-Application Communication**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_inter_application_communication)

[**32.1. Connecting Multiple Application Instances**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-connecting-multiple-application-instances)

[**32.2. Instance Index and Instance Count**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-instance-index-instance-count)

[**32.3. Partitioning**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-partitioning)

[32.3.1. Configuring Output Bindings for Partitioning](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-configuring-output-bindings-partitioning)

[32.3.2. Configuring Input Bindings for Partitioning](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-configuring-input-bindings-partitioning)

[**33. Testing**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_testing)

[**33.1. Disabling the Test Binder Autoconfiguration**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_disabling_the_test_binder_autoconfiguration)

[**34. Health Indicator**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_health_indicator_5)

[**35. Metrics Emitter**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-metrics-emitter)

[**36. Samples**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_samples)

[**36.1. Deploying Stream Applications on CloudFoundry**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_deploying_stream_applications_on_cloudfoundry)

[**VI. Binder Implementations**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_binder_implementations)

[**37. Apache Kafka Binder**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_apache_kafka_binder)

[**37.1. Usage**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_usage)

[**37.2. Apache Kafka Binder Overview**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_apache_kafka_binder_overview)

[**37.3. Configuration Options**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_configuration_options_2)

[37.3.1. Kafka Binder Properties](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_kafka_binder_properties)

[37.3.2. Kafka Consumer Properties](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#kafka-consumer-properties)

[37.3.3. Kafka Producer Properties](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#kafka-producer-properties)

[37.3.4. Usage examples](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_usage_examples)

[Example: Setting autoCommitOffset to false and Relying on Manual Acking](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_example_setting_literal_autocommitoffset_literal_to_literal_false_literal_and_relying_on_manual_acking)

[Example: Security Configuration](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_example_security_configuration)

[Example: Pausing and Resuming the Consumer](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#pause-resume)

[**37.4. Error Channels**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#kafka-error-channels)

[**37.5. Kafka Metrics**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#kafka-metrics)

[**37.6. Dead-Letter Topic Processing**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#kafka-dlq-processing)

[**37.7. Partitioning with the Kafka Binder**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_partitioning_with_the_kafka_binder)

[**38. Apache Kafka Streams Binder**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_apache_kafka_streams_binder)

[**38.1. Usage**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_usage_2)

[**38.2. Kafka Streams Binder Overview**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_kafka_streams_binder_overview)

[38.2.1. Streams DSL](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_streams_dsl)

[**38.3. Configuration Options**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_configuration_options_3)

[38.3.1. Kafka Streams Properties](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_kafka_streams_properties)

[38.3.2. TimeWindow properties:](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_timewindow_properties)

[**38.4. Multiple Input Bindings**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_multiple_input_bindings)

[38.4.1. Multiple Input Bindings as a Sink](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_multiple_input_bindings_as_a_sink)

[38.4.2. Multiple Input Bindings as a Processor](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_multiple_input_bindings_as_a_processor)

[**38.5. Multiple Output Bindings (aka Branching)**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_multiple_output_bindings_aka_branching)

[**38.6. Message Conversion**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_message_conversion)

[38.6.1. Outbound serialization](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_outbound_serialization)

[38.6.2. Inbound Deserialization](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_inbound_deserialization)

[**38.7. Error Handling**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_error_handling)

[38.7.1. Handling Deserialization Exceptions](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_handling_deserialization_exceptions)

[38.7.2. Handling Non-Deserialization Exceptions](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_handling_non_deserialization_exceptions)

[**38.8. Interactive Queries**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_interactive_queries)

[**38.9. Accessing the underlying KafkaStreams object**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_accessing_the_underlying_kafkastreams_object)

[**39. RabbitMQ Binder**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_rabbitmq_binder)

[**39.1. Usage**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_usage_3)

[**39.2. RabbitMQ Binder Overview**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_rabbitmq_binder_overview)

[**39.3. Configuration Options**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_configuration_options_4)

[39.3.1. RabbitMQ Binder Properties](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#rabbit-binder-properties)

[39.3.2. RabbitMQ Consumer Properties](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_rabbitmq_consumer_properties)

[39.3.3. Rabbit Producer Properties](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_rabbit_producer_properties)

[**39.4. Retry With the RabbitMQ Binder**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_retry_with_the_rabbitmq_binder)

[39.4.1. Putting it All Together](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_putting_it_all_together)

[**39.5. Error Channels**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#rabbit-error-channels)

[**39.6. Dead-Letter Queue Processing**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#rabbit-dlq-processing)

[39.6.1. Non-Partitioned Destinations](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_non_partitioned_destinations)

[39.6.2. Partitioned Destinations](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_partitioned_destinations)

[republishToDlq=false](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#__literal_republishtodlq_false_literal)

[republishToDlq=true](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#__literal_republishtodlq_true_literal)

[**39.7. Partitioning with the RabbitMQ Binder**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_partitioning_with_the_rabbitmq_binder)

[**VII. Spring Cloud Bus**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_cloud_bus)

[**40. Quick Start**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_quick_start_3)

[**41. Bus Endpoints**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_bus_endpoints)

[**41.1. Bus Refresh Endpoint**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_bus_refresh_endpoint)

[**41.2. Bus Env Endpoint**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_bus_env_endpoint)

[**42. Addressing an Instance**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_addressing_an_instance)

[**43. Addressing All Instances of a Service**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_addressing_all_instances_of_a_service)

[**44. Service ID Must Be Unique**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_service_id_must_be_unique)

[**45. Customizing the Message Broker**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_customizing_the_message_broker)

[**46. Tracing Bus Events**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_tracing_bus_events)

[**47. Broadcasting Your Own Events**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_broadcasting_your_own_events)

[**47.1. Registering events in custom packages**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_registering_events_in_custom_packages)

[**VIII. Spring Cloud Sleuth**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_cloud_sleuth)

[**48. Introduction**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_introduction)

[**48.1. Terminology**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_terminology)

[**48.2. Purpose**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_purpose)

[48.2.1. Distributed Tracing with Zipkin](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_distributed_tracing_with_zipkin)

[48.2.2. Visualizing errors](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_visualizing_errors)

[48.2.3. Distributed Tracing with Brave](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_distributed_tracing_with_brave)

[48.2.4. Live examples](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_live_examples)

[48.2.5. Log correlation](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_log_correlation)

[JSON Logback with Logstash](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_json_logback_with_logstash)

[48.2.6. Propagating Span Context](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_propagating_span_context)

[Baggage versus Span Tags](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_baggage_versus_span_tags)

[**48.3. Adding Sleuth to the Project**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#sleuth-adding-project)

[48.3.1. Only Sleuth (log correlation)](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_only_sleuth_log_correlation)

[48.3.2. Sleuth with Zipkin via HTTP](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_sleuth_with_zipkin_via_http)

[48.3.3. Sleuth with Zipkin over RabbitMQ or Kafka](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_sleuth_with_zipkin_over_rabbitmq_or_kafka)

[**49. Additional Resources**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_additional_resources)

[**50. Features**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_features_2)

[**50.1. Introduction to Brave**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_introduction_to_brave)

[50.1.1. Tracing](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_tracing)

[50.1.2. Local Tracing](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_local_tracing)

[50.1.3. Customizing Spans](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_customizing_spans)

[50.1.4. Implicitly Looking up the Current Span](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_implicitly_looking_up_the_current_span)

[50.1.5. RPC tracing](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_rpc_tracing)

[One-Way tracing](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_one_way_tracing)

[**51. Sampling**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_sampling)

[**51.1. Declarative sampling**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_declarative_sampling)

[**51.2. Custom sampling**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_custom_sampling)

[**51.3. Sampling in Spring Cloud Sleuth**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_sampling_in_spring_cloud_sleuth)

[**52. Propagation**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_propagation)

[**52.1. Propagating extra fields**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_propagating_extra_fields)

[52.1.1. Prefixed fields](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#prefixed-fields)

[52.1.2. Extracting a Propagated Context](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_extracting_a_propagated_context)

[52.1.3. Sharing span IDs between Client and Server](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_sharing_span_ids_between_client_and_server)

[52.1.4. Implementing Propagation](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_implementing_propagation)

[**53. Current Tracing Component**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_current_tracing_component)

[**54. Current Span**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_current_span)

[**54.1. Setting a span in scope manually**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_setting_a_span_in_scope_manually)

[**55. Instrumentation**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_instrumentation)

[**56. Span lifecycle**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_span_lifecycle)

[**56.1. Creating and finishing spans**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#creating-and-finishing-spans)

[**56.2. Continuing Spans**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#continuing-spans)

[**56.3. Creating a Span with an explicit Parent**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#creating-spans-with-explicit-parent)

[**57. Naming spans**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_naming_spans)

[**57.1. @SpanName Annotation**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#__literal_spanname_literal_annotation)

[**57.2. toString() method**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#__literal_tostring_literal_method)

[**58. Managing Spans with Annotations**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_managing_spans_with_annotations)

[**58.1. Rationale**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_rationale)

[**58.2. Creating New Spans**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_creating_new_spans)

[**58.3. Continuing Spans**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_continuing_spans)

[**58.4. Advanced Tag Setting**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_advanced_tag_setting)

[58.4.1. Custom extractor](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_custom_extractor)

[58.4.2. Resolving Expressions for a Value](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_resolving_expressions_for_a_value)

[58.4.3. Using the toString() method](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_using_the_literal_tostring_literal_method)

[**59. Customizations**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_customizations)

[**59.1. HTTP**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_http)

[**59.2. TracingFilter**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#__literal_tracingfilter_literal)

[**59.3. Custom service name**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_custom_service_name)

[**59.4. Customization of Reported Spans**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_customization_of_reported_spans)

[**59.5. Host Locator**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_host_locator)

[**60. Sending Spans to Zipkin**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_sending_spans_to_zipkin)

[**61. Zipkin Stream Span Consumer**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_zipkin_stream_span_consumer)

[**62. Integrations**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_integrations)

[**62.1. OpenTracing**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_opentracing)

[**62.2. Runnable and Callable**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_runnable_and_callable)

[**62.3. Hystrix**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_hystrix)

[62.3.1. Custom Concurrency Strategy](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_custom_concurrency_strategy)

[62.3.2. Manual Command setting](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_manual_command_setting)

[**62.4. RxJava**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_rxjava)

[**62.5. HTTP integration**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_http_integration)

[62.5.1. HTTP Filter](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_http_filter)

[62.5.2. HandlerInterceptor](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_handlerinterceptor)

[62.5.3. Async Servlet support](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_async_servlet_support)

[62.5.4. WebFlux support](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_webflux_support)

[62.5.5. Dubbo RPC support](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_dubbo_rpc_support)

[**62.6. HTTP Client Integration**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_http_client_integration)

[62.6.1. Synchronous Rest Template](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_synchronous_rest_template)

[62.6.2. Asynchronous Rest Template](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_asynchronous_rest_template)

[Multiple Asynchronous Rest Templates](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_multiple_asynchronous_rest_templates)

[62.6.3. WebClient](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#__literal_webclient_literal)

[62.6.4. Traverson](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_traverson)

[62.6.5. Apache HttpClientBuilder and HttpAsyncClientBuilder](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_apache_literal_httpclientbuilder_literal_and_literal_httpasyncclientbuilder_literal)

[62.6.6. Netty HttpClient](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_netty_literal_httpclient_literal)

[62.6.7. UserInfoRestTemplateCustomizer](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#__literal_userinforesttemplatecustomizer_literal)

[**62.7. Feign**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_feign)

[**62.8. Asynchronous Communication**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_asynchronous_communication)

[62.8.1. @Async Annotated methods](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#__literal_async_literal_annotated_methods)

[62.8.2. @Scheduled Annotated Methods](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#__literal_scheduled_literal_annotated_methods)

[62.8.3. Executor, ExecutorService, and ScheduledExecutorService](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_executor_executorservice_and_scheduledexecutorservice)

[Customization of Executors](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_customization_of_executors)

[**62.9. Messaging**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_messaging)

[62.9.1. Spring Integration and Spring Cloud Stream](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_integration_and_spring_cloud_stream)

[62.9.2. Spring RabbitMq](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_rabbitmq)

[62.9.3. Spring Kafka](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_kafka)

[**62.10. Zuul**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_zuul_2)

[**63. Running examples**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_running_examples)

[**IX. Spring Cloud Consul**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_cloud_consul)

[**64. Install Consul**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-consul-install)

[**65. Consul Agent**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-consul-agent)

[**66. Service Discovery with Consul**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-consul-discovery)

[**66.1. How to activate**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_how_to_activate)

[**66.2. Registering with Consul**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_registering_with_consul)

[**66.3. HTTP Health Check**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_http_health_check)

[66.3.1. Metadata and Consul tags](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_metadata_and_consul_tags)

[66.3.2. Making the Consul Instance ID Unique](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_making_the_consul_instance_id_unique)

[**66.4. Looking up services**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_looking_up_services)

[66.4.1. Using Ribbon](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_using_ribbon)

[66.4.2. Using the DiscoveryClient](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_using_the_discoveryclient)

[**66.5. Consul Catalog Watch**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_consul_catalog_watch)

[**67. Distributed Configuration with Consul**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-consul-config)

[**67.1. How to activate**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_how_to_activate_2)

[**67.2. Customizing**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_customizing)

[**67.3. Config Watch**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-consul-config-watch)

[**67.4. YAML or Properties with Config**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-consul-config-format)

[**67.5. git2consul with Config**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-consul-config-git2consul)

[**67.6. Fail Fast**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-consul-failfast)

[**68. Consul Retry**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-consul-retry)

[**69. Spring Cloud Bus with Consul**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-consul-bus)

[**69.1. How to activate**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_how_to_activate_3)

[**70. Circuit Breaker with Hystrix**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-consul-hystrix)

[**71. Hystrix metrics aggregation with Turbine and Consul**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-consul-turbine)

[**X. Spring Cloud Zookeeper**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_cloud_zookeeper)

[**72. Install Zookeeper**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-zookeeper-install)

[**73. Service Discovery with Zookeeper**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-zookeeper-discovery)

[**73.1. Activating**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_activating)

[**73.2. Registering with Zookeeper**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_registering_with_zookeeper)

[**73.3. Using the DiscoveryClient**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_using_the_discoveryclient_2)

[**74. Using Spring Cloud Zookeeper with Spring Cloud Netflix Components**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-zookeeper-netflix)

[**74.1. Ribbon with Zookeeper**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_ribbon_with_zookeeper)

[**75. Spring Cloud Zookeeper and Service Registry**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-zookeeper-service-registry)

[**75.1. Instance Status**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_instance_status)

[**76. Zookeeper Dependencies**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-zookeeper-dependencies)

[**76.1. Using the Zookeeper Dependencies**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-zookeeper-dependencies-using)

[**76.2. Activating Zookeeper Dependencies**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-zookeeper-dependencies-activating)

[**76.3. Setting up Zookeeper Dependencies**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-zookeeper-dependencies-setting-up)

[76.3.1. Aliases](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-zookeeper-dependencies-setting-up-aliases)

[76.3.2. Path](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_path)

[76.3.3. Load Balancer Type](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_load_balancer_type)

[76.3.4. Content-Type Template and Version](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#__literal_content_type_literal_template_and_version)

[76.3.5. Default Headers](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_default_headers)

[76.3.6. Required Dependencies](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_required_dependencies)

[76.3.7. Stubs](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_stubs)

[**76.4. Configuring Spring Cloud Zookeeper Dependencies**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-zookeeper-dependencies-configuring)

[**77. Spring Cloud Zookeeper Dependency Watcher**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-zookeeper-dependency-watcher)

[**77.1. Activating**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_activating_2)

[**77.2. Registering a Listener**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_registering_a_listener)

[**77.3. Using the Presence Checker**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-zookeeper-dependency-watcher-presence-checker)

[**78. Distributed Configuration with Zookeeper**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-zookeeper-config)

[**78.1. Activating**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_activating_3)

[**78.2. Customizing**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_customizing_2)

[**78.3. Access Control Lists (ACLs)**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_access_control_lists_acls)

[**XI. Spring Boot Cloud CLI**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_boot_cloud_cli)

[**79. Installation**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_installation)

[**80. Running Spring Cloud Services in Development**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_running_spring_cloud_services_in_development)

[**80.1. Adding Additional Applications**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_adding_additional_applications)

[**81. Writing Groovy Scripts and Running Applications**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_writing_groovy_scripts_and_running_applications)

[**82. Encryption and Decryption**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_encryption_and_decryption_3)

[**XII. Spring Cloud Security**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_cloud_security)

[**83. Quickstart**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_quickstart)

[**83.1. OAuth2 Single Sign On**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_oauth2_single_sign_on)

[**83.2. OAuth2 Protected Resource**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_oauth2_protected_resource)

[**84. More Detail**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_more_detail)

[**84.1. Single Sign On**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_single_sign_on)

[**84.2. Token Relay**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_token_relay)

[84.2.1. Client Token Relay](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_client_token_relay)

[84.2.2. Client Token Relay in Zuul Proxy](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_client_token_relay_in_zuul_proxy)

[84.2.3. Resource Server Token Relay](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_resource_server_token_relay)

[**85. Configuring Authentication Downstream of a Zuul Proxy**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_configuring_authentication_downstream_of_a_zuul_proxy)

[**XIII. Spring Cloud for Cloud Foundry**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_cloud_for_cloud_foundry)

[**86. Discovery**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_discovery)

[**87. Single Sign On**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_single_sign_on_2)

[**XIV. Spring Cloud Contract**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_cloud_contract)

[**88. Spring Cloud Contract**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_cloud_contract_2)

[**89. Spring Cloud Contract Verifier Introduction**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_cloud_contract_verifier_introduction)

[**89.1. Why a Contract Verifier?**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_why_a_contract_verifier)

[89.1.1. Testing issues](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_testing_issues)

[**89.2. Purposes**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_purposes)

[**89.3. How It Works**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_how_it_works)

[89.3.1. A Three-second Tour](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-contract-verifier-intro-three-second-tour)

[On the Producer Side](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-contract-verifier-intro-three-second-tour-producer)

[On the Consumer Side](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-contract-verifier-intro-three-second-tour-consumer)

[89.3.2. A Three-minute Tour](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-contract-verifier-intro-three-minute-tour)

[On the Producer Side](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-contract-verifier-intro-three-minute-tour-producer)

[On the Consumer Side](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-contract-verifier-intro-three-minute-tour-consumer)

[89.3.3. Defining the Contract](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_defining_the_contract)

[89.3.4. Client Side](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_client_side)

[89.3.5. Server Side](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_server_side)

[**89.4. Step-by-step Guide to Consumer Driven Contracts (CDC)**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_step_by_step_guide_to_consumer_driven_contracts_cdc)

[89.4.1. Technical note](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_technical_note)

[89.4.2. Consumer side (Loan Issuance)](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_consumer_side_loan_issuance)

[89.4.3. Producer side (Fraud Detection server)](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_producer_side_fraud_detection_server)

[89.4.4. Consumer Side (Loan Issuance) Final Step](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_consumer_side_loan_issuance_final_step)

[**89.5. Dependencies**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_dependencies)

[**89.6. Additional Links**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_additional_links)

[89.6.1. Spring Cloud Contract video](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_cloud_contract_video)

[89.6.2. Readings](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_readings)

[**89.7. Samples**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_samples_2)

[**90. Spring Cloud Contract FAQ**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_cloud_contract_faq)

[**90.1. Why use Spring Cloud Contract Verifier and not X ?**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_why_use_spring_cloud_contract_verifier_and_not_x)

[**90.2. I don’t want to write a contract in Groovy!**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_i_don_t_want_to_write_a_contract_in_groovy)

[**90.3. What is this value(consumer(), producer()) ?**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_what_is_this_value_consumer_producer)

[**90.4. How to do Stubs versioning?**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_how_to_do_stubs_versioning)

[90.4.1. API Versioning](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_api_versioning)

[90.4.2. JAR versioning](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_jar_versioning)

[90.4.3. Dev or prod stubs](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_dev_or_prod_stubs)

[**90.5. Common repo with contracts**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_common_repo_with_contracts)

[90.5.1. Repo structure](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_repo_structure)

[90.5.2. Workflow](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_workflow)

[90.5.3. Consumer](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_consumer)

[90.5.4. Producer](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_producer)

[90.5.5. How can I define messaging contracts per topic not per producer?](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_how_can_i_define_messaging_contracts_per_topic_not_per_producer)

[For Maven Project](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_for_maven_project)

[For Gradle Project](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_for_gradle_project)

[**90.6. Do I need a Binary Storage? Can’t I use Git?**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_do_i_need_a_binary_storage_can_t_i_use_git)

[90.6.1. Protocol convention](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_protocol_convention)

[90.6.2. Producer](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_producer_2)

[90.6.3. Consumer](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_consumer_2)

[**90.7. Can I use the Pact Broker?**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_can_i_use_the_pact_broker)

[90.7.1. Pact Consumer](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_pact_consumer)

[90.7.2. Producer](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_producer_3)

[90.7.3. Pact Consumer (Producer Contract approach)](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_pact_consumer_producer_contract_approach)

[**90.8. How can I debug the request/response being sent by the generated tests client?**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_how_can_i_debug_the_request_response_being_sent_by_the_generated_tests_client)

[90.8.1. How can I debug the mapping/request/response being sent by WireMock?](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_how_can_i_debug_the_mapping_request_response_being_sent_by_wiremock)

[90.8.2. How can I see what got registered in the HTTP server stub?](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_how_can_i_see_what_got_registered_in_the_http_server_stub)

[90.8.3. Can I reference text from file?](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_can_i_reference_text_from_file)

[**91. Spring Cloud Contract Verifier Setup**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_cloud_contract_verifier_setup)

[**91.1. Gradle Project**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gradle-project)

[91.1.1. Prerequisites](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gradle-prerequisites)

[91.1.2. Add Gradle Plugin with Dependencies](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gradle-add-gradle-plugin)

[91.1.3. Gradle and Rest Assured 2.0](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gradle-and-rest-assured)

[91.1.4. Snapshot Versions for Gradle](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gradle-snapshot-versions)

[91.1.5. Add stubs](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gradle-add-stubs)

[91.1.6. Run the Plugin](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gradle-run-plugin)

[91.1.7. Default Setup](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gradle-default-setup)

[91.1.8. Configure Plugin](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gradle-configure-plugin)

[91.1.9. Configuration Options](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gradle-configuration-options)

[91.1.10. Single Base Class for All Tests](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gradle-single-base-class)

[91.1.11. Different Base Classes for Contracts](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gradle-different-base-classes)

[91.1.12. Invoking Generated Tests](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gradle-invoking-generated-tests)

[91.1.13. Pushing stubs to SCM](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gradle-pushing-stubs-to-scm)

[91.1.14. Spring Cloud Contract Verifier on the Consumer Side](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gradle-consumer)

[**91.2. Maven Project**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#maven-project)

[91.2.1. Add maven plugin](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#maven-add-plugin)

[91.2.2. Maven and Rest Assured 2.0](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#maven-rest-assured)

[91.2.3. Snapshot versions for Maven](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#maven-snapshot-versions)

[91.2.4. Add stubs](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#maven-add-stubs)

[91.2.5. Run plugin](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#maven-run-plugin)

[91.2.6. Configure plugin](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#maven-configure-plugin)

[91.2.7. Configuration Options](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#maven-configuration-options)

[91.2.8. Single Base Class for All Tests](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#maven-single-base)

[91.2.9. Different base classes for contracts](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#maven-different-base)

[91.2.10. Invoking generated tests](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#maven-invoking-generated-tests)

[91.2.11. Pushing stubs to SCM](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#maven-pushing-stubs-to-scm)

[91.2.12. Maven Plugin and STS](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#maven-sts)

[**91.3. Stubs and Transitive Dependencies**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_stubs_and_transitive_dependencies)

[**91.4. CI Server setup**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_ci_server_setup)

[**91.5. Scenarios**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_scenarios)

[**91.6. Docker Project**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#docker-project)

[91.6.1. Short intro to Maven, JARs and Binary storage](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_short_intro_to_maven_jars_and_binary_storage)

[91.6.2. How it works](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_how_it_works_2)

[Environment Variables](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_environment_variables)

[91.6.3. Example of usage](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_example_of_usage)

[91.6.4. Server side (nodejs)](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#docker-server-side)

[**92. Spring Cloud Contract Verifier Messaging**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_cloud_contract_verifier_messaging)

[**92.1. Integrations**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_integrations_2)

[**92.2. Manual Integration Testing**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_manual_integration_testing)

[**92.3. Publisher-Side Test Generation**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_publisher_side_test_generation)

[92.3.1. Scenario 1: No Input Message](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_scenario_1_no_input_message)

[92.3.2. Scenario 2: Output Triggered by Input](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_scenario_2_output_triggered_by_input)

[92.3.3. Scenario 3: No Output Message](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_scenario_3_no_output_message)

[**92.4. Consumer Stub Generation**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_consumer_stub_generation)

[**93. Spring Cloud Contract Stub Runner**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_cloud_contract_stub_runner)

[**93.1. Snapshot versions**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_snapshot_versions)

[**93.2. Publishing Stubs as JARs**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#publishing-stubs-as-jars)

[**93.3. Stub Runner Core**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_stub_runner_core)

[93.3.1. Retrieving stubs](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_retrieving_stubs)

[Stub downloading](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_stub_downloading)

[Classpath scanning](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_classpath_scanning)

[93.3.2. Running stubs](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_running_stubs)

[Limitations](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_limitations)

[Running using main app](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_running_using_main_app)

[HTTP Stubs](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_http_stubs)

[Viewing registered mappings](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_viewing_registered_mappings)

[Messaging Stubs](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_messaging_stubs)

[**93.4. Stub Runner JUnit Rule**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_stub_runner_junit_rule)

[93.4.1. Maven settings](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_maven_settings)

[93.4.2. Providing fixed ports](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_providing_fixed_ports)

[93.4.3. Fluent API](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_fluent_api)

[93.4.4. Stub Runner with Spring](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_stub_runner_with_spring)

[**93.5. Stub Runner Spring Cloud**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_stub_runner_spring_cloud)

[93.5.1. Stubbing Service Discovery](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_stubbing_service_discovery)

[Test profiles and service discovery](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_test_profiles_and_service_discovery)

[93.5.2. Additional Configuration](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_additional_configuration)

[**93.6. Stub Runner Boot Application**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_stub_runner_boot_application)

[93.6.1. How to use it?](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_how_to_use_it)

[Stub Runner Server](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_stub_runner_server)

[Stub Runner Server Fat Jar](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_stub_runner_server_fat_jar)

[Spring Cloud CLI](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_cloud_cli)

[93.6.2. Endpoints](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_endpoints_2)

[HTTP](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_http_2)

[Messaging](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_messaging_2)

[93.6.3. Example](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_example)

[93.6.4. Stub Runner Boot with Service Discovery](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_stub_runner_boot_with_service_discovery)

[**93.7. Stubs Per Consumer**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_stubs_per_consumer)

[**93.8. Common**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_common)

[93.8.1. Common Properties for JUnit and Spring](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#common-properties-junit-spring)

[93.8.2. Stub Runner Stubs IDs](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#stub-runner-stub-ids)

[**93.9. Stub Runner Docker**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#stubrunner-docker)

[93.9.1. How to use it](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_how_to_use_it_2)

[93.9.2. Example of client side usage in a non JVM project](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_example_of_client_side_usage_in_a_non_jvm_project)

[**94. Stub Runner for Messaging**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_stub_runner_for_messaging)

[**94.1. Stub triggering**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_stub_triggering)

[94.1.1. Trigger by Label](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#trigger-label)

[94.1.2. Trigger by Group and Artifact Ids](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#trigger-group-artifact-ids)

[94.1.3. Trigger by Artifact Ids](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#trigger-artifact-ids)

[94.1.4. Trigger All Messages](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#trigger-all-messages)

[**94.2. Stub Runner Integration**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_stub_runner_integration)

[94.2.1. Adding the Runner to the Project](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_adding_the_runner_to_the_project)

[94.2.2. Disabling the functionality](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_disabling_the_functionality)

[Scenario 1 (no input message)](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#integration-scenario-1)

[Scenario 2 (output triggered by input)](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#integration-scenario-1)

[Scenario 3 (input with no output)](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#integration-scenario-3)

[**94.3. Stub Runner Stream**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_stub_runner_stream)

[94.3.1. Adding the Runner to the Project](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_adding_the_runner_to_the_project_2)

[94.3.2. Disabling the functionality](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_disabling_the_functionality_2)

[Scenario 1 (no input message)](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#stream-scenario-1)

[Scenario 2 (output triggered by input)](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#stream-scenario-2)

[Scenario 3 (input with no output)](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#stream-scenario-3)

[**94.4. Stub Runner Spring AMQP**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_stub_runner_spring_amqp)

[94.4.1. Adding the Runner to the Project](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_adding_the_runner_to_the_project_3)

[Triggering the message](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_triggering_the_message)

[Spring AMQP Test Configuration](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_amqp_test_configuration)

[**95. Contract DSL**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_contract_dsl)

[**95.1. Limitations**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_limitations_2)

[**95.2. Common Top-Level elements**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_common_top_level_elements)

[95.2.1. Description](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#contract-dsl-description)

[95.2.2. Name](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#contract-dsl-name)

[95.2.3. Ignoring Contracts](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#contract-dsl-ignoring-contracts)

[95.2.4. Passing Values from Files](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#contract-dsl-passing-values-from-files)

[95.2.5. HTTP Top-Level Elements](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#contract-dsl-http-top-level-elements)

[**95.3. Request**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_request)

[**95.4. Response**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_response)

[**95.5. Dynamic properties**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_dynamic_properties)

[95.5.1. Dynamic properties inside the body](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_dynamic_properties_inside_the_body)

[95.5.2. Regular expressions](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_regular_expressions)

[95.5.3. Passing Optional Parameters](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_passing_optional_parameters)

[95.5.4. Executing Custom Methods on the Server Side](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_executing_custom_methods_on_the_server_side)

[95.5.5. Referencing the Request from the Response](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_referencing_the_request_from_the_response)

[95.5.6. Registering Your Own WireMock Extension](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_registering_your_own_wiremock_extension)

[95.5.7. Dynamic Properties in the Matchers Sections](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#contract-matchers)

[**95.6. JAX-RS Support**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_jax_rs_support)

[**95.7. Async Support**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_async_support)

[**95.8. Working with Context Paths**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_working_with_context_paths)

[**95.9. Working with Web Flux**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_working_with_web_flux)

[**95.10. Messaging Top-Level Elements**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_messaging_top_level_elements)

[95.10.1. Output Triggered by a Method](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#contract-dsl-output-triggered-method)

[95.10.2. Output Triggered by a Message](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#contract-dsl-output-triggered-message)

[95.10.3. Consumer/Producer](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#contract-dsl-consumer-producer)

[95.10.4. Common](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#contract-dsl-common)

[**95.11. Multiple Contracts in One File**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_multiple_contracts_in_one_file)

[**95.12. Generating Spring REST Docs snippets from the contracts**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_generating_spring_rest_docs_snippets_from_the_contracts)

[**96. Customization**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_customization)

[**96.1. Extending the DSL**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_extending_the_dsl)

[96.1.1. Common JAR](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_common_jar)

[96.1.2. Adding the Dependency to the Project](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_adding_the_dependency_to_the_project)

[96.1.3. Test the Dependency in the Project’s Dependencies](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_test_the_dependency_in_the_project_s_dependencies)

[96.1.4. Test a Dependency in the Plugin’s Dependencies](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_test_a_dependency_in_the_plugin_s_dependencies)

[96.1.5. Referencing classes in DSLs](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_referencing_classes_in_dsls)

[**97. Using the Pluggable Architecture**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_using_the_pluggable_architecture)

[**97.1. Custom Contract Converter**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_custom_contract_converter)

[97.1.1. Pact Converter](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#pact-converter)

[97.1.2. Pact Contract](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_pact_contract)

[97.1.3. Pact for Producers](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_pact_for_producers)

[97.1.4. Pact for Consumers](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_pact_for_consumers)

[**97.2. Using the Custom Test Generator**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_using_the_custom_test_generator)

[**97.3. Using the Custom Stub Generator**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_using_the_custom_stub_generator)

[**97.4. Using the Custom Stub Runner**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_using_the_custom_stub_runner)

[**97.5. Using the Custom Stub Downloader**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_using_the_custom_stub_downloader)

[**97.6. Using the SCM Stub Downloader**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#scm-stub-downloader)

[**97.7. Using the Pact Stub Downloader**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#pact-stub-downloader)

[**98. Spring Cloud Contract WireMock**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_cloud_contract_wiremock)

[**98.1. Registering Stubs Automatically**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_registering_stubs_automatically)

[**98.2. Using Files to Specify the Stub Bodies**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_using_files_to_specify_the_stub_bodies)

[**98.3. Alternative: Using JUnit Rules**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_alternative_using_junit_rules)

[**98.4. Relaxed SSL Validation for Rest Template**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_relaxed_ssl_validation_for_rest_template)

[**98.5. WireMock and Spring MVC Mocks**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_wiremock_and_spring_mvc_mocks)

[**98.6. Customization of WireMock configuration**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_customization_of_wiremock_configuration)

[**98.7. Generating Stubs using REST Docs**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_generating_stubs_using_rest_docs)

[**98.8. Generating Contracts by Using REST Docs**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_generating_contracts_by_using_rest_docs)

[**99. Migrations**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_migrations)

[**99.1. 1.0.x → 1.1.x**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#cloud-verifier-1.0-1.1)

[99.1.1. New structure of generated stubs](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_new_structure_of_generated_stubs)

[**99.2. 1.1.x → 1.2.x**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#cloud-verifier-1.1-1.2)

[99.2.1. Custom HttpServerStub](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_custom_literal_httpserverstub_literal)

[99.2.2. New packages for generated tests](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_new_packages_for_generated_tests)

[99.2.3. New Methods in TemplateProcessor](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_new_methods_in_templateprocessor)

[99.2.4. RestAssured 3.0](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_restassured_3_0)

[**99.3. 1.2.x → 2.0.x**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#cloud-verifier-1.2-2.0)

[99.3.1. No Camel support](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_no_camel_support)

[**100. Links**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_links)

[**XV. Spring Cloud Vault**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_cloud_vault)

[**101. Quick Start**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_quick_start_4)

[**102. Client Side Usage**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_client_side_usage_2)

[**102.1. Authentication**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_authentication_2)

[**103. Authentication methods**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.authentication)

[**103.1. Token authentication**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.authentication.token)

[**103.2. AppId authentication**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.authentication.appid)

[103.2.1. Custom UserId](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_custom_userid)

[**103.3. AppRole authentication**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_approle_authentication)

[**103.4. AWS-EC2 authentication**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.authentication.awsec2)

[**103.5. AWS-IAM authentication**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.authentication.awsiam)

[**103.6. TLS certificate authentication**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.authentication.clientcert)

[**103.7. Cubbyhole authentication**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.authentication.cubbyhole)

[**103.8. Kubernetes authentication**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.authentication.kubernetes)

[**104. Secret Backends**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.backends)

[**104.1. Generic Backend**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.backends.generic)

[**104.2. Versioned Key-Value Backend**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.backends.kv.versioned)

[**104.3. Consul**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.backends.consul)

[**104.4. RabbitMQ**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.backends.rabbitmq)

[**104.5. AWS**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.backends.aws)

[**105. Database backends**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.backends.database-backends)

[**105.1. Database**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.backends.database)

[**105.2. Apache Cassandra**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.backends.cassandra)

[**105.3. MongoDB**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.backends.mongodb)

[**105.4. MySQL**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.backends.mysql)

[**105.5. PostgreSQL**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.backends.postgresql)

[**106. Configure PropertySourceLocator behavior**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.backends.configurer)

[**107. Service Registry Configuration**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_service_registry_configuration)

[**108. Vault Client Fail Fast**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.fail-fast)

[**109. Vault Client SSL configuration**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.ssl)

[**110. Lease lifecycle management (renewal and revocation)**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault-lease-renewal)

[**XVI. Spring Cloud Gateway**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_cloud_gateway)

[**111. How to Include Spring Cloud Gateway**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gateway-starter)

[**112. Glossary**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_glossary)

[**113. How It Works**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gateway-how-it-works)

[**114. Route Predicate Factories**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gateway-request-predicates-factories)

[**114.1. After Route Predicate Factory**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_after_route_predicate_factory)

[**114.2. Before Route Predicate Factory**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_before_route_predicate_factory)

[**114.3. Between Route Predicate Factory**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_between_route_predicate_factory)

[**114.4. Cookie Route Predicate Factory**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_cookie_route_predicate_factory)

[**114.5. Header Route Predicate Factory**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_header_route_predicate_factory)

[**114.6. Host Route Predicate Factory**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_host_route_predicate_factory)

[**114.7. Method Route Predicate Factory**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_method_route_predicate_factory)

[**114.8. Path Route Predicate Factory**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_path_route_predicate_factory)

[**114.9. Query Route Predicate Factory**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_query_route_predicate_factory)

[**114.10. RemoteAddr Route Predicate Factory**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_remoteaddr_route_predicate_factory)

[114.10.1. Modifying the way remote addresses are resolved](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_modifying_the_way_remote_addresses_are_resolved)

[**115. GatewayFilter Factories**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_gatewayfilter_factories)

[**115.1. AddRequestHeader GatewayFilter Factory**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_addrequestheader_gatewayfilter_factory)

[**115.2. AddRequestParameter GatewayFilter Factory**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_addrequestparameter_gatewayfilter_factory)

[**115.3. AddResponseHeader GatewayFilter Factory**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_addresponseheader_gatewayfilter_factory)

[**115.4. Hystrix GatewayFilter Factory**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_hystrix_gatewayfilter_factory)

[**115.5. PrefixPath GatewayFilter Factory**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_prefixpath_gatewayfilter_factory)

[**115.6. PreserveHostHeader GatewayFilter Factory**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_preservehostheader_gatewayfilter_factory)

[**115.7. RequestRateLimiter GatewayFilter Factory**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_requestratelimiter_gatewayfilter_factory)

[115.7.1. Redis RateLimiter](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_redis_ratelimiter)

[**115.8. RedirectTo GatewayFilter Factory**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_redirectto_gatewayfilter_factory)

[**115.9. RemoveNonProxyHeaders GatewayFilter Factory**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_removenonproxyheaders_gatewayfilter_factory)

[**115.10. RemoveRequestHeader GatewayFilter Factory**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_removerequestheader_gatewayfilter_factory)

[**115.11. RemoveResponseHeader GatewayFilter Factory**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_removeresponseheader_gatewayfilter_factory)

[**115.12. RewritePath GatewayFilter Factory**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_rewritepath_gatewayfilter_factory)

[**115.13. SaveSession GatewayFilter Factory**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_savesession_gatewayfilter_factory)

[**115.14. SecureHeaders GatewayFilter Factory**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_secureheaders_gatewayfilter_factory)

[**115.15. SetPath GatewayFilter Factory**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_setpath_gatewayfilter_factory)

[**115.16. SetResponseHeader GatewayFilter Factory**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_setresponseheader_gatewayfilter_factory)

[**115.17. SetStatus GatewayFilter Factory**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_setstatus_gatewayfilter_factory)

[**115.18. StripPrefix GatewayFilter Factory**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_stripprefix_gatewayfilter_factory)

[**115.19. Retry GatewayFilter Factory**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_retry_gatewayfilter_factory)

[**116. Global Filters**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_global_filters)

[**116.1. Combined Global Filter and GatewayFilter Ordering**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_combined_global_filter_and_gatewayfilter_ordering)

[**116.2. Forward Routing Filter**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_forward_routing_filter)

[**116.3. LoadBalancerClient Filter**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_loadbalancerclient_filter)

[**116.4. Netty Routing Filter**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_netty_routing_filter)

[**116.5. Netty Write Response Filter**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_netty_write_response_filter)

[**116.6. RouteToRequestUrl Filter**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_routetorequesturl_filter)

[**116.7. Websocket Routing Filter**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_websocket_routing_filter)

[**116.8. Gateway Metrics Filter**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_gateway_metrics_filter)

[**116.9. Making An Exchange As Routed**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_making_an_exchange_as_routed)

[**117. TLS / SSL**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_tls_ssl)

[**118. Configuration**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_configuration_2)

[**118.1. Fluent Java Routes API**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_fluent_java_routes_api)

[**118.2. DiscoveryClient Route Definition Locator**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_discoveryclient_route_definition_locator)

[**119. CORS Configuration**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_cors_configuration)

[**120. Actuator API**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_actuator_api)

[**121. Developer Guide**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_developer_guide)

[**121.1. Writing Custom Route Predicate Factories**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_writing_custom_route_predicate_factories)

[**121.2. Writing Custom GatewayFilter Factories**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_writing_custom_gatewayfilter_factories)

[**121.3. Writing Custom Global Filters**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_writing_custom_global_filters)

[**121.4. Writing Custom Route Locators and Writers**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_writing_custom_route_locators_and_writers)

[**122. Building a Simple Gateway Using Spring MVC or Webflux**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_building_a_simple_gateway_using_spring_mvc_or_webflux)

[**XVII. Spring Cloud Function**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_spring_cloud_function)

[**123. Introduction**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_introduction_2)

[**124. Getting Started**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_getting_started)

[**125. Building and Running a Function**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_building_and_running_a_function)

[**126. Function Catalog and Flexible Function Signatures**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_function_catalog_and_flexible_function_signatures)

[**127. Standalone Web Applications**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_standalone_web_applications)

[**128. Standalone Streaming Applications**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_standalone_streaming_applications)

[**129. Deploying a Packaged Function**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_deploying_a_packaged_function)

[**130. Dynamic Compilation**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_dynamic_compilation)

[**131. Serverless Platform Adapters**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_serverless_platform_adapters)

[**131.1. AWS Lambda**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_aws_lambda)

[131.1.1. Introduction](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_introduction_3)

[131.1.2. Notes on JAR Layout](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_notes_on_jar_layout)

[131.1.3. Upload](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_upload)

[131.1.4. Platfom Specific Features](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_platfom_specific_features)

[HTTP and API Gateway](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_http_and_api_gateway)

[**131.2. Azure Functions**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_azure_functions)

[131.2.1. Notes on JAR Layout](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_notes_on_jar_layout_2)

[131.2.2. JSON Configuration](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_json_configuration)

[131.2.3. Build](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_build)

[131.2.4. Running the sample](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_running_the_sample)

[**131.3. Apache Openwhisk**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_apache_openwhisk)

[131.3.1. Quick Start](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_quick_start_5)

[**XVIII. Appendix: Compendium of Configuration Properties**](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_appendix_compendium_of_configuration_properties)

Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus). Coordination of distributed systems leads to boiler plate patterns, and using Spring Cloud developers can quickly stand up services and applications that implement those patterns. They will work well in any distributed environment, including the developer’s own laptop, bare metal data centres, and managed platforms such as Cloud Foundry.

Version: Finchley.SR1

**1. Features**

Spring Cloud focuses on providing good out of box experience for typical use cases and extensibility mechanism to cover others.

* Distributed/versioned configuration
* Service registration and discovery
* Routing
* Service-to-service calls
* Load balancing
* Circuit Breakers
* Distributed messaging

**Part I. Cloud Native Applications**

[Cloud Native](https://pivotal.io/platform-as-a-service/migrating-to-cloud-native-application-architectures-ebook) is a style of application development that encourages easy adoption of best practices in the areas of continuous delivery and value-driven development. A related discipline is that of building [12-factor Applications](http://12factor.net/), in which development practices are aligned with delivery and operations goals — for instance, by using declarative programming and management and monitoring. Spring Cloud facilitates these styles of development in a number of specific ways. The starting point is a set of features to which all components in a distributed system need easy access.

Many of those features are covered by [Spring Boot](https://projects.spring.io/spring-boot), on which Spring Cloud builds. Some more features are delivered by Spring Cloud as two libraries: Spring Cloud Context and Spring Cloud Commons. Spring Cloud Context provides utilities and special services for the ApplicationContext of a Spring Cloud application (bootstrap context, encryption, refresh scope, and environment endpoints). Spring Cloud Commons is a set of abstractions and common classes used in different Spring Cloud implementations (such as Spring Cloud Netflix and Spring Cloud Consul).

If you get an exception due to "Illegal key size" and you use Sun’s JDK, you need to install the Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files. See the following links for more information:

* [Java 6 JCE](http://www.oracle.com/technetwork/java/javase/downloads/jce-6-download-429243.html)
* [Java 7 JCE](http://www.oracle.com/technetwork/java/javase/downloads/jce-7-download-432124.html)
* [Java 8 JCE](http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html)

Extract the files into the JDK/jre/lib/security folder for whichever version of JRE/JDK x64/x86 you use.

|  |
| --- |
| [Note] |
| Spring Cloud is released under the non-restrictive Apache 2.0 license. If you would like to contribute to this section of the documentation or if you find an error, you can find the source code and issue trackers for the project at [github](https://github.com/spring-cloud/spring-cloud-commons/tree/master/docs/src/main/asciidoc). |

**2. Spring Cloud Context: Application Context Services**

Spring Boot has an opinionated view of how to build an application with Spring. For instance, it has conventional locations for common configuration files and has endpoints for common management and monitoring tasks. Spring Cloud builds on top of that and adds a few features that probably all components in a system would use or occasionally need.

**2.1 The Bootstrap Application Context**

A Spring Cloud application operates by creating a “bootstrap” context, which is a parent context for the main application. It is responsible for loading configuration properties from the external sources and for decrypting properties in the local external configuration files. The two contexts share an Environment, which is the source of external properties for any Spring application. By default, bootstrap properties (not bootstrap.properties but properties that are loaded during the bootstrap phase) are added with high precedence, so they cannot be overridden by local configuration.

The bootstrap context uses a different convention for locating external configuration than the main application context. Instead of application.yml (or .properties), you can use bootstrap.yml, keeping the external configuration for bootstrap and main context nicely separate. The following listing shows an example:

**bootstrap.yml.**

spring:

application:

name: foo

cloud:

config:

uri: ${SPRING\_CONFIG\_URI:http://localhost:8888}

If your application needs any application-specific configuration from the server, it is a good idea to set the spring.application.name (in bootstrap.yml or application.yml).

You can disable the bootstrap process completely by setting spring.cloud.bootstrap.enabled=false (for example, in system properties).

**2.2 Application Context Hierarchies**

If you build an application context from SpringApplication or SpringApplicationBuilder, then the Bootstrap context is added as a parent to that context. It is a feature of Spring that child contexts inherit property sources and profiles from their parent, so the “main” application context contains additional property sources, compared to building the same context without Spring Cloud Config. The additional property sources are:

* “bootstrap”: If any PropertySourceLocators are found in the Bootstrap context and if they have non-empty properties, an optional CompositePropertySourceappears with high priority. An example would be properties from the Spring Cloud Config Server. See “[Section 2.6, “Customizing the Bootstrap Property Sources”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#customizing-bootstrap-property-sources)” for instructions on how to customize the contents of this property source.
* “applicationConfig: [classpath:bootstrap.yml]” (and related files if Spring profiles are active): If you have a bootstrap.yml (or .properties), those properties are used to configure the Bootstrap context. Then they get added to the child context when its parent is set. They have lower precedence than the application.yml(or .properties) and any other property sources that are added to the child as a normal part of the process of creating a Spring Boot application. See “[Section 2.3, “Changing the Location of Bootstrap Properties”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#customizing-bootstrap-properties)” for instructions on how to customize the contents of these property sources.

Because of the ordering rules of property sources, the “bootstrap” entries take precedence. However, note that these do not contain any data from bootstrap.yml, which has very low precedence but can be used to set defaults.

You can extend the context hierarchy by setting the parent context of any ApplicationContext you create — for example, by using its own interface or with the SpringApplicationBuilder convenience methods (parent(), child() and sibling()). The bootstrap context is the parent of the most senior ancestor that you create yourself. Every context in the hierarchy has its own “bootstrap” (possibly empty) property source to avoid promoting values inadvertently from parents down to their descendants. If there is a Config Server, every context in the hierarchy can also (in principle) have a different spring.application.name and, hence, a different remote property source. Normal Spring application context behavior rules apply to property resolution: properties from a child context override those in the parent, by name and also by property source name. (If the child has a property source with the same name as the parent, the value from the parent is not included in the child).

Note that the SpringApplicationBuilder lets you share an Environment amongst the whole hierarchy, but that is not the default. Thus, sibling contexts, in particular, do not need to have the same profiles or property sources, even though they may share common values with their parent.

**2.3 Changing the Location of Bootstrap Properties**

The bootstrap.yml (or .properties) location can be specified by setting spring.cloud.bootstrap.name (default: bootstrap) or spring.cloud.bootstrap.location (default: empty) — for example, in System properties. Those properties behave like the spring.config.\* variants with the same name. In fact, they are used to set up the bootstrap ApplicationContext by setting those properties in its Environment. If there is an active profile (from spring.profiles.active or through the Environment API in the context you are building), properties in that profile get loaded as well, the same as in a regular Spring Boot app — for example, from bootstrap-development.properties for a development profile.

**2.4 Overriding the Values of Remote Properties**

The property sources that are added to your application by the bootstrap context are often “remote” (from example, from Spring Cloud Config Server). By default, they cannot be overridden locally. If you want to let your applications override the remote properties with their own System properties or config files, the remote property source has to grant it permission by setting spring.cloud.config.allowOverride=true (it does not work to set this locally). Once that flag is set, two finer-grained settings control the location of the remote properties in relation to system properties and the application’s local configuration:

* spring.cloud.config.overrideNone=true: Override from any local property source.
* spring.cloud.config.overrideSystemProperties=false: Only system properties, command line arguments, and environment variables (but not the local config files) should override the remote settings.

**2.5 Customizing the Bootstrap Configuration**

The bootstrap context can be set to do anything you like by adding entries to /META-INF/spring.factories under a key named org.springframework.cloud.bootstrap.BootstrapConfiguration. This holds a comma-separated list of Spring @Configuration classes that are used to create the context. Any beans that you want to be available to the main application context for autowiring can be created here. There is a special contract for @Beans of type ApplicationContextInitializer. If you want to control the startup sequence, classes can be marked with an @Order annotation (the default order is last).

|  |
| --- |
| [Warning] |
| When adding custom BootstrapConfiguration, be careful that the classes you add are not @ComponentScanned by mistake into your “main” application context, where they might not be needed. Use a separate package name for boot configuration classes and make sure that name is not already covered by your @ComponentScan or @SpringBootApplication annotated configuration classes. |

The bootstrap process ends by injecting initializers into the main SpringApplication instance (which is the normal Spring Boot startup sequence, whether it is running as a standalone application or deployed in an application server). First, a bootstrap context is created from the classes found in spring.factories. Then, all @Beansof type ApplicationContextInitializer are added to the main SpringApplication before it is started.

**2.6 Customizing the Bootstrap Property Sources**

The default property source for external configuration added by the bootstrap process is the Spring Cloud Config Server, but you can add additional sources by adding beans of type PropertySourceLocator to the bootstrap context (through spring.factories). For instance, you can insert additional properties from a different server or from a database.

As an example, consider the following custom locator:

*@Configuration*

**public** **class** CustomPropertySourceLocator **implements** PropertySourceLocator {

*@Override*

**public** PropertySource<?> locate(Environment environment) {

**return** **new** MapPropertySource("customProperty",

Collections.<String, Object>singletonMap("property.from.sample.custom.source", "worked as intended"));

}

}

The Environment that is passed in is the one for the ApplicationContext about to be created — in other words, the one for which we supply additional property sources for. It already has its normal Spring Boot-provided property sources, so you can use those to locate a property source specific to this Environment (for example, by keying it on spring.application.name, as is done in the default Spring Cloud Config Server property source locator).

If you create a jar with this class in it and then add a META-INF/spring.factories containing the following, the customProperty PropertySource appears in any application that includes that jar on its classpath:

org.springframework.cloud.bootstrap.BootstrapConfiguration=sample.custom.CustomPropertySourceLocator

**2.7 Logging Configuration**

If you are going to use Spring Boot to configure log settings than you should place this configuration in `bootstrap.[yml | properties] if you would like it to apply to all events.

|  |
| --- |
| [Note] |
| For Spring Cloud to initialize logging configuration properly you cannot use a custom prefix. For example, using custom.loggin.logpath will not be recognized by Spring Cloud when initializing the logging system. |

**2.8 Environment Changes**

The application listens for an EnvironmentChangeEvent and reacts to the change in a couple of standard ways (additional ApplicationListeners can be added as @Beans by the user in the normal way). When an EnvironmentChangeEvent is observed, it has a list of key values that have changed, and the application uses those to:

* Re-bind any @ConfigurationProperties beans in the context
* Set the logger levels for any properties in logging.level.\*

Note that the Config Client does not, by default, poll for changes in the Environment. Generally, we would not recommend that approach for detecting changes (although you could set it up with a @Scheduled annotation). If you have a scaled-out client application, it is better to broadcast the EnvironmentChangeEvent to all the instances instead of having them polling for changes (for example, by using the [Spring Cloud Bus](https://github.com/spring-cloud/spring-cloud-bus)).

The EnvironmentChangeEvent covers a large class of refresh use cases, as long as you can actually make a change to the Environment and publish the event. Note that those APIs are public and part of core Spring). You can verify that the changes are bound to @ConfigurationProperties beans by visiting the /configpropsendpoint (a normal Spring Boot Actuator feature). For instance, a DataSource can have its maxPoolSize changed at runtime (the default DataSource created by Spring Boot is an @ConfigurationProperties bean) and grow capacity dynamically. Re-binding @ConfigurationProperties does not cover another large class of use cases, where you need more control over the refresh and where you need a change to be atomic over the whole ApplicationContext. To address those concerns, we have @RefreshScope.

**2.9 Refresh Scope**

When there is a configuration change, a Spring @Bean that is marked as @RefreshScope gets special treatment. This feature addresses the problem of stateful beans that only get their configuration injected when they are initialized. For instance, if a DataSource has open connections when the database URL is changed via the Environment, you probably want the holders of those connections to be able to complete what they are doing. Then, the next time something borrows a connection from the pool, it gets one with the new URL.

Sometimes, it might even be mandatory to apply the @RefreshScope annotation on some beans which can be only initialized once. If a bean is "immutable", you will have to either annotate the bean with @RefreshScope or specify the classname under the property key spring.cloud.refresh.extra-refreshable.

Refresh scope beans are lazy proxies that initialize when they are used (that is, when a method is called), and the scope acts as a cache of initialized values. To force a bean to re-initialize on the next method call, you must invalidate its cache entry.

The RefreshScope is a bean in the context and has a public refreshAll() method to refresh all beans in the scope by clearing the target cache. The /refreshendpoint exposes this functionality (over HTTP or JMX). To refresh an individual bean by name, there is also a refresh(String) method.

To expose the /refresh endpoint, you need to add following configuration to your application:

management:

endpoints:

web:

exposure:

include: refresh

|  |
| --- |
| [Note] |
| @RefreshScope works (technically) on an @Configuration class, but it might lead to surprising behavior. For example, it does not mean that all the @Beans defined in that class are themselves in @RefreshScope. Specifically, anything that depends on those beans cannot rely on them being updated when a refresh is initiated, unless it is itself in @RefreshScope. In that case, it is rebuilt on a refresh and its dependencies are re-injected. At that point, they are re-initialized from the refreshed @Configuration). |

**2.10 Encryption and Decryption**

Spring Cloud has an Environment pre-processor for decrypting property values locally. It follows the same rules as the Config Server and has the same external configuration through encrypt.\*. Thus, you can use encrypted values in the form of {cipher}\* and, as long as there is a valid key, they are decrypted before the main application context gets the Environment settings. To use the encryption features in an application, you need to include Spring Security RSA in your classpath (Maven co-ordinates: "org.springframework.security:spring-security-rsa"), and you also need the full strength JCE extensions in your JVM.

If you get an exception due to "Illegal key size" and you use Sun’s JDK, you need to install the Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files. See the following links for more information:

* [Java 6 JCE](http://www.oracle.com/technetwork/java/javase/downloads/jce-6-download-429243.html)
* [Java 7 JCE](http://www.oracle.com/technetwork/java/javase/downloads/jce-7-download-432124.html)
* [Java 8 JCE](http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html)

Extract the files into the JDK/jre/lib/security folder for whichever version of JRE/JDK x64/x86 you use.

**2.11 Endpoints**

For a Spring Boot Actuator application, some additional management endpoints are available. You can use:

* POST to /actuator/env to update the Environment and rebind @ConfigurationProperties and log levels.
* /actuator/refresh to re-load the boot strap context and refresh the @RefreshScope beans.
* /actuator/restart to close the ApplicationContext and restart it (disabled by default).
* /actuator/pause and /actuator/resume for calling the Lifecycle methods (stop() and start() on the ApplicationContext).

|  |
| --- |
| [Note] |
| If you disable the /actuator/restart endpoint then the /actuator/pause and /actuator/resume endpoints will also be disabled since they are just a special case of /actuator/restart. |

**3. Spring Cloud Commons: Common Abstractions**

Patterns such as service discovery, load balancing, and circuit breakers lend themselves to a common abstraction layer that can be consumed by all Spring Cloud clients, independent of the implementation (for example, discovery with Eureka or Consul).

**3.1 @EnableDiscoveryClient**

Spring Cloud Commons provides the @EnableDiscoveryClient annotation. This looks for implementations of the DiscoveryClient interface with META-INF/spring.factories. Implementations of the Discovery Client add a configuration class to spring.factories under the org.springframework.cloud.client.discovery.EnableDiscoveryClient key. Examples of DiscoveryClient implementations include [Spring Cloud Netflix Eureka](https://cloud.spring.io/spring-cloud-netflix/), [Spring Cloud Consul Discovery](https://cloud.spring.io/spring-cloud-consul/), and [Spring Cloud Zookeeper Discovery](https://cloud.spring.io/spring-cloud-zookeeper/).

By default, implementations of DiscoveryClient auto-register the local Spring Boot server with the remote discovery server. This behavior can be disabled by setting autoRegister=false in @EnableDiscoveryClient.

|  |
| --- |
| [Note] |
| @EnableDiscoveryClient is no longer required. You can put a DiscoveryClient implementation on the classpath to cause the Spring Boot application to register with the service discovery server. |

**3.1.1 Health Indicator**

Commons creates a Spring Boot HealthIndicator that DiscoveryClient implementations can participate in by implementing DiscoveryHealthIndicator. To disable the composite HealthIndicator, set spring.cloud.discovery.client.composite-indicator.enabled=false. A generic HealthIndicator based on DiscoveryClient is auto-configured (DiscoveryClientHealthIndicator). To disable it, set spring.cloud.discovery.client.health-indicator.enabled=false. To disable the description field of the DiscoveryClientHealthIndicator, set spring.cloud.discovery.client.health-indicator.include-description=false. Otherwise, it can bubble up as the description of the rolled up HealthIndicator.

**3.2 ServiceRegistry**

Commons now provides a ServiceRegistry interface that provides methods such as register(Registration) and deregister(Registration), which let you provide custom registered services. Registration is a marker interface.

The following example shows the ServiceRegistry in use:

*@Configuration*

*@EnableDiscoveryClient(autoRegister=false)*

**public** **class** MyConfiguration {

**private** ServiceRegistry registry;

**public** MyConfiguration(ServiceRegistry registry) {

**this**.registry = registry;

}

*// called through some external process, such as an event or a custom actuator endpoint*

**public** **void** register() {

Registration registration = constructRegistration();

**this**.registry.register(registration);

}

}

Each ServiceRegistry implementation has its own Registry implementation.

* ZookeeperRegistration used with ZookeeperServiceRegistry
* EurekaRegistration used with EurekaServiceRegistry
* ConsulRegistration used with ConsulServiceRegistry

If you are using the ServiceRegistry interface, you are going to need to pass the correct Registry implementation for the ServiceRegistry implementation you are using.

**3.2.1 ServiceRegistry Auto-Registration**

By default, the ServiceRegistry implementation auto-registers the running service. To disable that behavior, you can set: \* @EnableDiscoveryClient(autoRegister=false) to permanently disable auto-registration. \* spring.cloud.service-registry.auto-registration.enabled=false to disable the behavior through configuration.

**3.2.2 Service Registry Actuator Endpoint**

Spring Cloud Commons provides a /service-registry actuator endpoint. This endpoint relies on a Registration bean in the Spring Application Context. Calling /service-registry with GET returns the status of the Registration. Using POST to the same endpoint with a JSON body changes the status of the current Registration to the new value. The JSON body has to include the status field with the preferred value. Please see the documentation of the ServiceRegistryimplementation you use for the allowed values when updating the status and the values returned for the status. For instance, Eureka’s supported statuses are UP, DOWN, OUT\_OF\_SERVICE, and UNKNOWN.

**3.3 Spring RestTemplate as a Load Balancer Client**

RestTemplate can be automatically configured to use ribbon. To create a load-balanced RestTemplate, create a RestTemplate @Bean and use the @LoadBalanced qualifier, as shown in the following example:

*@Configuration*

**public** **class** MyConfiguration {

*@LoadBalanced*

*@Bean*

RestTemplate restTemplate() {

**return** **new** RestTemplate();

}

}

**public** **class** MyClass {

*@Autowired*

**private** RestTemplate restTemplate;

**public** String doOtherStuff() {

String results = restTemplate.getForObject("http://stores/stores", String.**class**);

**return** results;

}

}

|  |  |
| --- | --- |
| [Caution] | **Caution** |
| A RestTemplate bean is no longer created through auto-configuration. Individual applications must create it. |

The URI needs to use a virtual host name (that is, a service name, not a host name). The Ribbon client is used to create a full physical address. See [RibbonAutoConfiguration](https://github.com/spring-cloud/spring-cloud-netflix/blob/master/spring-cloud-netflix-core/src/main/java/org/springframework/cloud/netflix/ribbon/RibbonAutoConfiguration.java) for details of how the RestTemplate is set up.

**3.4 Spring WebClient as a Load Balancer Client**

WebClient can be automatically configured to use the LoadBalancerClient. To create a load-balanced WebClient, create a WebClient.Builder @Bean and use the @LoadBalanced qualifier, as shown in the following example:

*@Configuration*

**public** **class** MyConfiguration {

*@Bean*

*@LoadBalanced*

**public** WebClient.Builder loadBalancedWebClientBuilder() {

**return** WebClient.builder();

}

}

**public** **class** MyClass {

*@Autowired*

**private** WebClient.Builder webClientBuilder;

**public** Mono<String> doOtherStuff() {

**return** webClientBuilder.build().get().uri("http://stores/stores")

.retrieve().bodyToMono(String.**class**);

}

}

The URI needs to use a virtual host name (that is, a service name, not a host name). The Ribbon client is used to create a full physical address.

**3.4.1 Retrying Failed Requests**

A load-balanced RestTemplate can be configured to retry failed requests. By default, this logic is disabled. You can enable it by adding [Spring Retry](https://github.com/spring-projects/spring-retry) to your application’s classpath. The load-balanced RestTemplate honors some of the Ribbon configuration values related to retrying failed requests. You can use client.ribbon.MaxAutoRetries, client.ribbon.MaxAutoRetriesNextServer, and client.ribbon.OkToRetryOnAllOperations properties. If you would like to disable the retry logic with Spring Retry on the classpath, you can set spring.cloud.loadbalancer.retry.enabled=false. See the [Ribbon documentation](https://github.com/Netflix/ribbon/wiki/Getting-Started#the-properties-file-sample-clientproperties) for a description of what these properties do.

If you would like to implement a BackOffPolicy in your retries, you need to create a bean of type LoadBalancedBackOffPolicyFactory and return the BackOffPolicy you would like to use for a given service, as shown in the following example:

*@Configuration*

**public** **class** MyConfiguration {

*@Bean*

LoadBalancedBackOffPolicyFactory backOffPolciyFactory() {

**return** **new** LoadBalancedBackOffPolicyFactory() {

*@Override*

**public** BackOffPolicy createBackOffPolicy(String service) {

**return** **new** ExponentialBackOffPolicy();

}

};

}

}

|  |
| --- |
| [Note] |
| client in the preceding examples should be replaced with your Ribbon client’s name. |

If you want to add one or more RetryListener implementations to your retry functionality, you need to create a bean of type LoadBalancedRetryListenerFactoryand return the RetryListener array you would like to use for a given service, as shown in the following example:

*@Configuration*

**public** **class** MyConfiguration {

*@Bean*

LoadBalancedRetryListenerFactory retryListenerFactory() {

**return** **new** LoadBalancedRetryListenerFactory() {

*@Override*

**public** RetryListener[] createRetryListeners(String service) {

**return** **new** RetryListener[]{**new** RetryListener() {

*@Override*

**public** <T, E **extends** Throwable> **boolean** open(RetryContext context, RetryCallback<T, E> callback) {

*//TODO Do you business...*

**return** true;

}

*@Override*

**public** <T, E **extends** Throwable> **void** close(RetryContext context, RetryCallback<T, E> callback, Throwable throwable) {

*//TODO Do you business...*

}

*@Override*

**public** <T, E **extends** Throwable> **void** onError(RetryContext context, RetryCallback<T, E> callback, Throwable throwable) {

*//TODO Do you business...*

}

}};

}

};

}

}

**3.5 Multiple RestTemplate objects**

If you want a RestTemplate that is not load-balanced, create a RestTemplate bean and inject it. To access the load-balanced RestTemplate, use the @LoadBalanced qualifier when you create your @Bean, as shown in the following example:\

*@Configuration*

**public** **class** MyConfiguration {

*@LoadBalanced*

*@Bean*

RestTemplate loadBalanced() {

**return** **new** RestTemplate();

}

*@Primary*

*@Bean*

RestTemplate restTemplate() {

**return** **new** RestTemplate();

}

}

**public** **class** MyClass {

*@Autowired*

**private** RestTemplate restTemplate;

*@Autowired*

*@LoadBalanced*

**private** RestTemplate loadBalanced;

**public** String doOtherStuff() {

**return** loadBalanced.getForObject("http://stores/stores", String.**class**);

}

**public** String doStuff() {

**return** restTemplate.getForObject("http://example.com", String.**class**);

}

}

|  |  |
| --- | --- |
| [Important] | **Important** |
| Notice the use of the @Primary annotation on the plain RestTemplate declaration in the preceding example to disambiguate the unqualified @Autowired injection. |
| [Tip] |
| If you see errors such as java.lang.IllegalArgumentException: Can not set org.springframework.web.client.RestTemplate field com.my.app.Foo.restTemplate to com.sun.proxy.$Proxy89, try injecting RestOperations or setting spring.aop.proxyTargetClass=true. | |

**3.6 Spring WebFlux WebClient as a Load Balancer Client**

WebClient can be configured to use the LoadBalancerClient. LoadBalancerExchangeFilterFunction is auto-configured if spring-webflux is on the classpath. The following example shows how to configure a WebClient to use load balancer:

**public** **class** MyClass {

*@Autowired*

**private** LoadBalancerExchangeFilterFunction lbFunction;

**public** Mono<String> doOtherStuff() {

**return** WebClient.builder().baseUrl("http://stores")

.filter(lbFunction)

.build()

.get()

.uri("/stores")

.retrieve()

.bodyToMono(String.**class**);

}

}

The URI needs to use a virtual host name (that is, a service name, not a host name). The LoadBalancerClient is used to create a full physical address.

**3.7 Ignore Network Interfaces**

Sometimes, it is useful to ignore certain named network interfaces so that they can be excluded from Service Discovery registration (for example, when running in a Docker container). A list of regular expressions can be set to cause the desired network interfaces to be ignored. The following configuration ignores the docker0interface and all interfaces that start with veth:

**application.yml.**

spring:

cloud:

inetutils:

ignoredInterfaces:

- docker0

- veth.\*

You can also force the use of only specified network addresses by using a list of regular expressions, as shown in the following example:

**bootstrap.yml.**

spring:

cloud:

inetutils:

preferredNetworks:

- 192.168

- 10.0

You can also force the use of only site-local addresses, as shown in the following example: .application.yml

spring:

cloud:

inetutils:

useOnlySiteLocalInterfaces: true

See [Inet4Address.html.isSiteLocalAddress()](https://docs.oracle.com/javase/8/docs/api/java/net/Inet4Address.html#isSiteLocalAddress--) for more details about what constitutes a site-local address.

**3.8 HTTP Client Factories**

Spring Cloud Commons provides beans for creating both Apache HTTP clients (ApacheHttpClientFactory) and OK HTTP clients (OkHttpClientFactory). The OkHttpClientFactory bean is created only if the OK HTTP jar is on the classpath. In addition, Spring Cloud Commons provides beans for creating the connection managers used by both clients: ApacheHttpClientConnectionManagerFactory for the Apache HTTP client and OkHttpClientConnectionPoolFactory for the OK HTTP client. If you would like to customize how the HTTP clients are created in downstream projects, you can provide your own implementation of these beans. In addition, if you provide a bean of type HttpClientBuilder or OkHttpClient.Builder, the default factories use these builders as the basis for the builders returned to downstream projects. You can also disable the creation of these beans by setting spring.cloud.httpclientfactories.apache.enabled or spring.cloud.httpclientfactories.ok.enabled to false.

**3.9 Enabled Features**

Spring Cloud Commons provides a /features actuator endpoint. This endpoint returns features available on the classpath and whether they are enabled. The information returned includes the feature type, name, version, and vendor.

**3.9.1 Feature types**

There are two types of 'features': abstract and named.

Abstract features are features where an interface or abstract class is defined and that an implementation the creates, such as DiscoveryClient, LoadBalancerClient, or LockService. The abstract class or interface is used to find a bean of that type in the context. The version displayed is bean.getClass().getPackage().getImplementationVersion().

Named features are features that do not have a particular class they implement, such as "Circuit Breaker", "API Gateway", "Spring Cloud Bus", and others. These features require a name and a bean type.

**3.9.2 Declaring features**

Any module can declare any number of HasFeature beans, as shown in the following examples:

*@Bean*

**public** HasFeatures commonsFeatures() {

**return** HasFeatures.abstractFeatures(DiscoveryClient.**class**, LoadBalancerClient.**class**);

}

*@Bean*

**public** HasFeatures consulFeatures() {

**return** HasFeatures.namedFeatures(

**new** NamedFeature("Spring Cloud Bus", ConsulBusAutoConfiguration.**class**),

**new** NamedFeature("Circuit Breaker", HystrixCommandAspect.**class**));

}

*@Bean*

HasFeatures localFeatures() {

**return** HasFeatures.builder()

.abstractFeature(Foo.**class**)

.namedFeature(**new** NamedFeature("Bar Feature", Bar.**class**))

.abstractFeature(Baz.**class**)

.build();

}

Each of these beans should go in an appropriately guarded @Configuration.

**Part II. Spring Cloud Config**

**Finchley.SR1**

Spring Cloud Config provides server-side and client-side support for externalized configuration in a distributed system. With the Config Server, you have a central place to manage external properties for applications across all environments. The concepts on both client and server map identically to the Spring Environment and PropertySource abstractions, so they fit very well with Spring applications but can be used with any application running in any language. As an application moves through the deployment pipeline from dev to test and into production, you can manage the configuration between those environments and be certain that applications have everything they need to run when they migrate. The default implementation of the server storage backend uses git, so it easily supports labelled versions of configuration environments as well as being accessible to a wide range of tooling for managing the content. It is easy to add alternative implementations and plug them in with Spring configuration.

**4. Quick Start**

This quick start walks through using both the server and the client of Spring Cloud Config Server.

First, start the server, as follows:

$ cd spring-cloud-config-server

$ ../mvnw spring-boot:run

The server is a Spring Boot application, so you can run it from your IDE if you prefer to do so (the main class is ConfigServerApplication).

Next try out a client, as follows:

$ curl localhost:8888/foo/development

{"name":"foo","label":"master","propertySources":[

{"name":"https://github.com/scratches/config-repo/foo-development.properties","source":{"bar":"spam"}},

{"name":"https://github.com/scratches/config-repo/foo.properties","source":{"foo":"bar"}}

]}

The default strategy for locating property sources is to clone a git repository (at spring.cloud.config.server.git.uri) and use it to initialize a mini SpringApplication. The mini-application’s Environment is used to enumerate property sources and publish them at a JSON endpoint.

The HTTP service has resources in the following form:

/{application}/{profile}[/{label}]

/{application}-{profile}.yml

/{label}/{application}-{profile}.yml

/{application}-{profile}.properties

/{label}/{application}-{profile}.properties

where application is injected as the spring.config.name in the SpringApplication (what is normally application in a regular Spring Boot app), profile is an active profile (or comma-separated list of properties), and label is an optional git label (defaults to master.)

Spring Cloud Config Server pulls configuration for remote clients from a git repository (which must be provided), as shown in the following example:

spring:

cloud:

config:

server:

git:

uri: https://github.com/spring-cloud-samples/config-repo

**4.1 Client Side Usage**

To use these features in an application, you can build it as a Spring Boot application that depends on spring-cloud-config-client (for an example, see the test cases for the config-client or the sample application). The most convenient way to add the dependency is with a Spring Boot starter org.springframework.cloud:spring-cloud-starter-config. There is also a parent pom and BOM (spring-cloud-starter-parent) for Maven users and a Spring IO version management properties file for Gradle and Spring CLI users. The following example shows a typical Maven configuration:

**pom.xml.**

<parent>

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

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

<version>{spring-boot-docs-version}</version>

<relativePath /> *<!-- lookup parent from repository -->*

</parent>

<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>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

<dependency>

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

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

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

*<!-- repositories also needed for snapshots and milestones -->*

Now you can create a standard Spring Boot application, such as the following HTTP server:

@SpringBootApplication

@RestController

public class Application {

@RequestMapping("/")

public String home() {

return "Hello World!";

}

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

When this HTTP server runs, it picks up the external configuration from the default local config server (if it is running) on port 8888. To modify the startup behavior, you can change the location of the config server by using bootstrap.properties (similar to application.properties but for the bootstrap phase of an application context), as shown in the following example:

spring.cloud.config.uri: http://myconfigserver.com

The bootstrap properties show up in the /env endpoint as a high-priority property source, as shown in the following example.

$ curl localhost:8080/env

{

"profiles":[],

"configService:https://github.com/spring-cloud-samples/config-repo/bar.properties":{"foo":"bar"},

"servletContextInitParams":{},

"systemProperties":{...},

...

}

A property source called ``configService:<URL of remote repository>/<file name> contains the foo property with a value of bar and is highest priority.

|  |
| --- |
| [Note] |
| The URL in the property source name is the git repository, not the config server URL. |

**5. Spring Cloud Config Server**

Spring Cloud Config Server provides an HTTP resource-based API for external configuration (name-value pairs or equivalent YAML content). The server is embeddable in a Spring Boot application, by using the @EnableConfigServer annotation. Consequently, the following application is a config server:

**ConfigServer.java.**

*@SpringBootApplication*

*@EnableConfigServer*

**public** **class** ConfigServer {

**public** **static** **void** main(String[] args) {

SpringApplication.run(ConfigServer.**class**, args);

}

}

Like all Spring Boot applications, it runs on port 8080 by default, but you can switch it to the more conventional port 8888 in various ways. The easiest, which also sets a default configuration repository, is by launching it with spring.config.name=configserver (there is a configserver.yml in the Config Server jar). Another is to use your own application.properties, as shown in the following example:

**application.properties.**

server.port: 8888

spring.cloud.config.server.git.uri: file://${user.home}/config-repo

where ${user.home}/config-repo is a git repository containing YAML and properties files.

|  |
| --- |
| [Note] |
| On Windows, you need an extra "/" in the file URL if it is absolute with a drive prefix (for example,file:///${user.home}/config-repo). |
| [Tip] |
| The following listing shows a recipe for creating the git repository in the preceding example:  $ cd $HOME  $ mkdir config-repo  $ cd config-repo  $ git init .  $ echo info.foo: bar > application.properties  $ git add -A .  $ git commit -m "Add application.properties" | |

|  |
| --- |
| [Warning] |
| Using the local filesystem for your git repository is intended for testing only. You should use a server to host your configuration repositories in production. |
| [Warning] |
| The initial clone of your configuration repository can be quick and efficient if you keep only text files in it. If you store binary files, especially large ones, you may experience delays on the first request for configuration or encounter out of memory errors in the server. | |

**5.1 Environment Repository**

Where should you store the configuration data for the Config Server? The strategy that governs this behaviour is the EnvironmentRepository, serving Environmentobjects. This Environment is a shallow copy of the domain from the Spring Environment (including propertySources as the main feature). The Environmentresources are parametrized by three variables:

* {application}, which maps to spring.application.name on the client side.
* {profile}, which maps to spring.profiles.active on the client (comma-separated list).
* {label}, which is a server side feature labelling a "versioned" set of config files.

Repository implementations generally behave like a Spring Boot application, loading configuration files from a spring.config.name equal to the {application}parameter, and spring.profiles.active equal to the {profiles} parameter. Precedence rules for profiles are also the same as in a regular Spring Boot application: Active profiles take precedence over defaults, and, if there are multiple profiles, the last one wins (similar to adding entries to a Map).

The following sample client application has this bootstrap configuration:

**bootstrap.yml.**

spring:

application:

name: foo

profiles:

active: dev,mysql

(As usual with a Spring Boot application, these properties could also be set by environment variables or command line arguments).

If the repository is file-based, the server creates an Environment from application.yml (shared between all clients) and foo.yml (with foo.yml taking precedence). If the YAML files have documents inside them that point to Spring profiles, those are applied with higher precedence (in order of the profiles listed). If there are profile-specific YAML (or properties) files, these are also applied with higher precedence than the defaults. Higher precedence translates to a PropertySource listed earlier in the Environment. (These same rules apply in a standalone Spring Boot application.)

You can set spring.cloud.config.server.accept-empty to false so that Server would return a HTTP 404 status, if the application is not found.By default, this flag is set to true.

**5.1.1 Git Backend**

The default implementation of EnvironmentRepository uses a Git backend, which is very convenient for managing upgrades and physical environments and for auditing changes. To change the location of the repository, you can set the spring.cloud.config.server.git.uri configuration property in the Config Server (for example in application.yml). If you set it with a file: prefix, it should work from a local repository so that you can get started quickly and easily without a server. However, in that case, the server operates directly on the local repository without cloning it (it does not matter if it is not bare because the Config Server never makes changes to the "remote" repository). To scale the Config Server up and make it highly available, you need to have all instances of the server pointing to the same repository, so only a shared file system would work. Even in that case, it is better to use the ssh: protocol for a shared filesystem repository, so that the server can clone it and use a local working copy as a cache.

This repository implementation maps the {label} parameter of the HTTP resource to a git label (commit id, branch name, or tag). If the git branch or tag name contains a slash (/), then the label in the HTTP URL should instead be specified with the special string (\_) (to avoid ambiguity with other URL paths). For example, if the label is foo/bar, replacing the slash would result in the following label: foo(\_)bar. The inclusion of the special string (\_) can also be applied to the {application}parameter. If you use a command-line client such as curl, be careful with the brackets in the URL — you should escape them from the shell with single quotes ('').

**Skipping SSL Certificate Validation**

The configuration server’s validation of the Git server’s SSL certificate can be disabled by setting the git.skipSslValidation property to true (default is false).

spring:

cloud:

config:

server:

git:

uri: https://example.com/my/repo

skipSslValidation: **true**

**Setting HTTP Connection Timeout**

You can configure the time, in seconds, that the configuration server will wait to acquire an HTTP connection. Use the git.timeout property.

spring:

cloud:

config:

server:

git:

uri: https://example.com/my/repo

timeout: 4

**Placeholders in Git URI**

Spring Cloud Config Server supports a git repository URL with placeholders for the {application} and {profile} (and {label} if you need it, but remember that the label is applied as a git label anyway). So you can support a “one repository per application” policy by using a structure similar to the following:

spring:

cloud:

config:

server:

git:

uri: https://github.com/myorg/{application**}**

You can also support a “one repository per profile” policy by using a similar pattern but with {profile}.

Additionally, using the special string "(\_)" within your {application} parameters can enable support for multiple organizations, as shown in the following example:

spring:

cloud:

config:

server:

git:

uri: https://github.com/{application**}**

where {application} is provided at request time in the following format: organization(\_)application.

**Pattern Matching and Multiple Repositories**

Spring Cloud Config also includes support for more complex requirements with pattern matching on the application and profile name. The pattern format is a comma-separated list of {application}/{profile} names with wildcards (note that a pattern beginning with a wildcard may need to be quoted), as shown in the following example:

spring:

cloud:

config:

server:

git:

uri: https://github.com/spring-cloud-samples/config-repo

repos:

simple: https://github.com/simple/config-repo

special:

pattern: special\*/dev\*,\*special\*/dev\*

uri: https://github.com/special/config-repo

local:

pattern: local\*

uri: file:/home/configsvc/config-repo

If {application}/{profile} does not match any of the patterns, it uses the default URI defined under spring.cloud.config.server.git.uri. In the above example, for the “simple” repository, the pattern is simple/\* (it only matches one application named simple in all profiles). The “local” repository matches all application names beginning with local in all profiles (the /\* suffix is added automatically to any pattern that does not have a profile matcher).

|  |
| --- |
| [Note] |
| The “one-liner” short cut used in the “simple” example can be used only if the only property to be set is the URI. If you need to set anything else (credentials, pattern, and so on) you need to use the full form. |

The pattern property in the repo is actually an array, so you can use a YAML array (or [0], [1], etc. suffixes in properties files) to bind to multiple patterns. You may need to do so if you are going to run apps with multiple profiles, as shown in the following example:

spring:

cloud:

config:

server:

git:

uri: https://github.com/spring-cloud-samples/config-repo

repos:

development:

pattern:

- '\*/development'

- '\*/staging'

uri: https://github.com/development/config-repo

staging:

pattern:

- '\*/qa'

- '\*/production'

uri: https://github.com/staging/config-repo

|  |
| --- |
| [Note] |
| Spring Cloud guesses that a pattern containing a profile that does not end in \* implies that you actually want to match a list of profiles starting with this pattern (so \*/staging is a shortcut for ["\*/staging", "\*/staging,\*"], and so on). This is common where, for instance, you need to run applications in the “development” profile locally but also the “cloud” profile remotely. |

Every repository can also optionally store config files in sub-directories, and patterns to search for those directories can be specified as searchPaths. The following example shows a config file at the top level:

spring:

cloud:

config:

server:

git:

uri: https://github.com/spring-cloud-samples/config-repo

searchPaths: foo,bar\*

In the preceding example, the server searches for config files in the top level and in the foo/ sub-directory and also any sub-directory whose name begins with bar.

By default, the server clones remote repositories when configuration is first requested. The server can be configured to clone the repositories at startup, as shown in the following top-level example:

spring:

cloud:

config:

server:

git:

uri: https://git/common/config-repo.git

repos:

team-a:

pattern: team-a-\*

cloneOnStart: **true**

uri: http://git/team-a/config-repo.git

team-b:

pattern: team-b-\*

cloneOnStart: **false**

uri: http://git/team-b/config-repo.git

team-c:

pattern: team-c-\*

uri: http://git/team-a/config-repo.git

In the preceding example, the server clones team-a’s config-repo on startup, before it accepts any requests. All other repositories are not cloned until configuration from the repository is requested.

|  |
| --- |
| [Note] |
| Setting a repository to be cloned when the Config Server starts up can help to identify a misconfigured configuration source (such as an invalid repository URI) quickly, while the Config Server is starting up. With cloneOnStart not enabled for a configuration source, the Config Server may start successfully with a misconfigured or invalid configuration source and not detect an error until an application requests configuration from that configuration source. |

**Authentication**

To use HTTP basic authentication on the remote repository, add the username and password properties separately (not in the URL), as shown in the following example:

spring:

cloud:

config:

server:

git:

uri: https://github.com/spring-cloud-samples/config-repo

username: trolley

password: strongpassword

If you do not use HTTPS and user credentials, SSH should also work out of the box when you store keys in the default directories (~/.ssh) and the URI points to an SSH location, such as git@github.com:configuration/cloud-configuration. It is important that an entry for the Git server be present in the ~/.ssh/known\_hosts file and that it is in ssh-rsa format. Other formats (such as ecdsa-sha2-nistp256) are not supported. To avoid surprises, you should ensure that only one entry is present in the known\_hosts file for the Git server and that it matches the URL you provided to the config server. If you use a hostname in the URL, you want to have exactly that (not the IP) in the known\_hosts file. The repository is accessed by using JGit, so any documentation you find on that should be applicable. HTTPS proxy settings can be set in ~/.git/config or (in the same way as for any other JVM process) with system properties (-Dhttps.proxyHost and -Dhttps.proxyPort).

|  |
| --- |
| [Tip] |
| If you do not know where your ~/.git directory is, use git config --global to manipulate the settings (for example, git config --global http.sslVerify false). |

**Authentication with AWS CodeCommit**

Spring Cloud Config Server also supports [AWS CodeCommit](https://docs.aws.amazon.com/codecommit/latest/userguide/welcome.html) authentication. AWS CodeCommit uses an authentication helper when using Git from the command line. This helper is not used with the JGit library, so a JGit CredentialProvider for AWS CodeCommit is created if the Git URI matches the AWS CodeCommit pattern. AWS CodeCommit URIs follow this pattern://git-codecommit.${AWS\_REGION}.amazonaws.com/${repopath}.

If you provide a username and password with an AWS CodeCommit URI, they must be the [AWS accessKeyId and secretAccessKey](https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSGettingStartedGuide/AWSCredentials.html) that provide access to the repository. If you do not specify a username and password, the accessKeyId and secretAccessKey are retrieved by using the [AWS Default Credential Provider Chain](https://docs.aws.amazon.com/sdk-for-java/v1/developer-guide/credentials.html).

If your Git URI matches the CodeCommit URI pattern (shown earlier), you must provide valid AWS credentials in the username and password or in one of the locations supported by the default credential provider chain. AWS EC2 instances may use [IAM Roles for EC2 Instances](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/iam-roles-for-amazon-ec2.html).

|  |
| --- |
| [Note] |
| The aws-java-sdk-core jar is an optional dependency. If the aws-java-sdk-core jar is not on your classpath, the AWS Code Commit credential provider is not created, regardless of the git server URI. |

**Git SSH configuration using properties**

By default, the JGit library used by Spring Cloud Config Server uses SSH configuration files such as ~/.ssh/known\_hosts and /etc/ssh/ssh\_config when connecting to Git repositories by using an SSH URI. In cloud environments such as Cloud Foundry, the local filesystem may be ephemeral or not easily accessible. For those cases, SSH configuration can be set by using Java properties. In order to activate property-based SSH configuration, the spring.cloud.config.server.git.ignoreLocalSshSettings property must be set to true, as shown in the following example:

spring:

cloud:

config:

server:

git:

uri: git*@gitserver.com:team/repo1.git*

ignoreLocalSshSettings: **true**

hostKey: someHostKey

hostKeyAlgorithm: ssh-rsa

privateKey: |

-----BEGIN RSA PRIVATE KEY-----

MIIEpgIBAAKCAQEAx4UbaDzY5xjW6hc9jwN0mX33XpTDVW9WqHp5AKaRbtAC3DqX

IXFMPgw3K45jxRb93f8tv9vL3rD9CUG1Gv4FM+o7ds7FRES5RTjv2RT/JVNJCoqF

ol8+ngLqRZCyBtQN7zYByWMRirPGoDUqdPYrj2yq+ObBBNhg5N+hOwKjjpzdj2Ud

1l7R+wxIqmJo1IYyy16xS8WsjyQuyC0lL456qkd5BDZ0Ag8j2X9H9D5220Ln7s9i

oezTipXipS7p7Jekf3Ywx6abJwOmB0rX79dV4qiNcGgzATnG1PkXxqt76VhcGa0W

DDVHEEYGbSQ6hIGSh0I7BQun0aLRZojfE3gqHQIDAQABAoIBAQCZmGrk8BK6tXCd

fY6yTiKxFzwb38IQP0ojIUWNrq0+9Xt+NsypviLHkXfXXCKKU4zUHeIGVRq5MN9b

BO56/RrcQHHOoJdUWuOV2qMqJvPUtC0CpGkD+valhfD75MxoXU7s3FK7yjxy3rsG

EmfA6tHV8/4a5umo5TqSd2YTm5B19AhRqiuUVI1wTB41DjULUGiMYrnYrhzQlVvj

5MjnKTlYu3V8PoYDfv1GmxPPh6vlpafXEeEYN8VB97e5x3DGHjZ5UrurAmTLTdO8

+AahyoKsIY612TkkQthJlt7FJAwnCGMgY6podzzvzICLFmmTXYiZ/28I4BX/mOSe

pZVnfRixAoGBAO6Uiwt40/PKs53mCEWngslSCsh9oGAaLTf/XdvMns5VmuyyAyKG

ti8Ol5wqBMi4GIUzjbgUvSUt+IowIrG3f5tN85wpjQ1UGVcpTnl5Qo9xaS1PFScQ

xrtWZ9eNj2TsIAMp/svJsyGG3OibxfnuAIpSXNQiJPwRlW3irzpGgVx/AoGBANYW

dnhshUcEHMJi3aXwR12OTDnaLoanVGLwLnkqLSYUZA7ZegpKq90UAuBdcEfgdpyi

PhKpeaeIiAaNnFo8m9aoTKr+7I6/uMTlwrVnfrsVTZv3orxjwQV20YIBCVRKD1uX

VhE0ozPZxwwKSPAFocpyWpGHGreGF1AIYBE9UBtjAoGBAI8bfPgJpyFyMiGBjO6z

FwlJc/xlFqDusrcHL7abW5qq0L4v3R+FrJw3ZYufzLTVcKfdj6GelwJJO+8wBm+R

gTKYJItEhT48duLIfTDyIpHGVm9+I1MGhh5zKuCqIhxIYr9jHloBB7kRm0rPvYY4

VAykcNgyDvtAVODP+4m6JvhjAoGBALbtTqErKN47V0+JJpapLnF0KxGrqeGIjIRV

cYA6V4WYGr7NeIfesecfOC356PyhgPfpcVyEztwlvwTKb3RzIT1TZN8fH4YBr6Ee

KTbTjefRFhVUjQqnucAvfGi29f+9oE3Ei9f7wA+H35ocF6JvTYUsHNMIO/3gZ38N

CPjyCMa9AoGBAMhsITNe3QcbsXAbdUR00dDsIFVROzyFJ2m40i4KCRM35bC/BIBs

q0TY3we+ERB40U8Z2BvU61QuwaunJ2+uGadHo58VSVdggqAo0BSkH58innKKt96J

69pcVH/4rmLbXdcmNYGm6iu+MlPQk4BUZknHSmVHIFdJ0EPupVaQ8RHT

-----END RSA PRIVATE KEY-----

The following table describes the SSH configuration properties.

**Table 5.1. SSH Configuration Properties**

| **Property Name** | **Remarks** |
| --- | --- |
| **ignoreLocalSshSettings** | If true, use property-based instead of file-based SSH config. Must be set at as spring.cloud.config.server.git.ignoreLocalSshSettings, **not** inside a repository definition. |
| **privateKey** | Valid SSH private key. Must be set if ignoreLocalSshSettings is true and Git URI is SSH format. |
| **hostKey** | Valid SSH host key. Must be set if hostKeyAlgorithm is also set. |
| **hostKeyAlgorithm** | One of ssh-dss, ssh-rsa, ecdsa-sha2-nistp256, ecdsa-sha2-nistp384, or ecdsa-sha2-nistp521. Must be set if hostKey is also set. |
| **strictHostKeyChecking** | true or false. If false, ignore errors with host key. |
| **knownHostsFile** | Location of custom .known\_hosts file. |
| **preferredAuthentications** | Override server authentication method order. This should allow for evading login prompts if server has keyboard-interactive authentication before the publickey method. |

**Placeholders in Git Search Paths**

Spring Cloud Config Server also supports a search path with placeholders for the {application} and {profile} (and {label} if you need it), as shown in the following example:

spring:

cloud:

config:

server:

git:

uri: https://github.com/spring-cloud-samples/config-repo

searchPaths: '{application}'

The preceding listing causes a search of the repository for files in the same name as the directory (as well as the top level). Wildcards are also valid in a search path with placeholders (any matching directory is included in the search).

**Force pull in Git Repositories**

As mentioned earlier, Spring Cloud Config Server makes a clone of the remote git repository in case the local copy gets dirty (for example, folder content changes by an OS process) such that Spring Cloud Config Server cannot update the local copy from remote repository.

To solve this issue, there is a force-pull property that makes Spring Cloud Config Server force pull from the remote repository if the local copy is dirty, as shown in the following example:

spring:

cloud:

config:

server:

git:

uri: https://github.com/spring-cloud-samples/config-repo

force-pull: **true**

If you have a multiple-repositories configuration, you can configure the force-pull property per repository, as shown in the following example:

spring:

cloud:

config:

server:

git:

uri: https://git/common/config-repo.git

force-pull: **true**

repos:

team-a:

pattern: team-a-\*

uri: http://git/team-a/config-repo.git

force-pull: **true**

team-b:

pattern: team-b-\*

uri: http://git/team-b/config-repo.git

force-pull: **true**

team-c:

pattern: team-c-\*

uri: http://git/team-a/config-repo.git

|  |
| --- |
| [Note] |
| The default value for force-pull property is false. |

**Deleting untracked branches in Git Repositories**

As Spring Cloud Config Server has a clone of the remote git repository after check-outing branch to local repo (e.g fetching properties by label) it will keep this branch forever or till the next server restart (which creates new local repo). So there could be a case when remote branch is deleted but local copy of it is still available for fetching. And if Spring Cloud Config Server client service starts with --spring.cloud.config.label=deletedRemoteBranch,master it will fetch properties from deletedRemoteBranch local branch, but not from master.

In order to keep local repository branches clean and up to remote - deleteUntrackedBranches property could be set. It will make Spring Cloud Config Server **force**delete untracked branches from local repository. Example:

spring:

cloud:

config:

server:

git:

uri: https://github.com/spring-cloud-samples/config-repo

deleteUntrackedBranches: **true**

|  |
| --- |
| [Note] |
| The default value for deleteUntrackedBranches property is false. |

**5.1.2 Version Control Backend Filesystem Use**

|  |
| --- |
| [Warning] |
| With VCS-based backends (git, svn), files are checked out or cloned to the local filesystem. By default, they are put in the system temporary directory with a prefix of config-repo-. On linux, for example, it could be /tmp/config-repo-<randomid>. Some operating systems [routinely clean out](https://serverfault.com/questions/377348/when-does-tmp-get-cleared/377349#377349) temporary directories. This can lead to unexpected behavior, such as missing properties. To avoid this problem, change the directory that Config Server uses by setting spring.cloud.config.server.git.basedir or spring.cloud.config.server.svn.basedir to a directory that does not reside in the system temp structure. |

**5.1.3 File System Backend**

There is also a “native” profile in the Config Server that does not use Git but loads the config files from the local classpath or file system (any static URL you want to point to with spring.cloud.config.server.native.searchLocations). To use the native profile, launch the Config Server with spring.profiles.active=native.

|  |
| --- |
| [Note] |
| Remember to use the file: prefix for file resources (the default without a prefix is usually the classpath). As with any Spring Boot configuration, you can embed ${}-style environment placeholders, but remember that absolute paths in Windows require an extra / (for example, file:///${user.home}/config-repo). |
| [Warning] |
| The default value of the searchLocations is identical to a local Spring Boot application (that is, [classpath:/, classpath:/config, file:./, file:./config]). This does not expose the application.properties from the server to all clients, because any property sources present in the server are removed before being sent to the client. | |

|  |
| --- |
| [Tip] |
| A filesystem backend is great for getting started quickly and for testing. To use it in production, you need to be sure that the file system is reliable and shared across all instances of the Config Server. |

The search locations can contain placeholders for {application}, {profile}, and {label}. In this way, you can segregate the directories in the path and choose a strategy that makes sense for you (such as subdirectory per application or subdirectory per profile).

If you do not use placeholders in the search locations, this repository also appends the {label} parameter of the HTTP resource to a suffix on the search path, so properties files are loaded from each search location **and** a subdirectory with the same name as the label (the labelled properties take precedence in the Spring Environment). Thus, the default behaviour with no placeholders is the same as adding a search location ending with /{label}/. For example, file:/tmp/config is the same as file:/tmp/config,file:/tmp/config/{label}. This behavior can be disabled by setting spring.cloud.config.server.native.addLabelLocations=false.

**5.1.4 Vault Backend**

Spring Cloud Config Server also supports [Vault](https://www.vaultproject.io/) as a backend.

Vault is a tool for securely accessing secrets. A secret is anything that to which you want to tightly control access, such as API keys, passwords, certificates, and other sensitive information. Vault provides a unified interface to any secret while providing tight access control and recording a detailed audit log.

For more information on Vault, see the [Vault quick start guide](https://www.vaultproject.io/intro/index.html).

To enable the config server to use a Vault backend, you can run your config server with the vault profile. For example, in your config server’s application.properties, you can add spring.profiles.active=vault.

By default, the config server assumes that your Vault server runs at [http://127.0.0.1:8200](http://127.0.0.1:8200/). It also assumes that the name of backend is secret and the key is application. All of these defaults can be configured in your config server’s application.properties. The following table describes configurable Vault properties:

| **Name** | | **Default Value** |
| --- | --- | --- |
| host | | 127.0.0.1 |
| port | | 8200 |
| scheme | | http |
| backend | | secret |
| defaultKey | | application |
| profileSeparator | | , |
| kvVersion | | 1 |
| skipSslValidation | | false |
| timeout | | 5 |
| [Important] | **Important** |
| All of the properties in the preceding table must be prefixed with spring.cloud.config.server.vault. |

All configurable properties can be found in org.springframework.cloud.config.server.environment.VaultEnvironmentRepository.

Vault 0.10.0 introduced a versioned key-value backend (k/v backend version 2) that exposes a different API than earlier versions, it now requires a data/ between the mount path and the actual context path and wraps secrets in a data object. Setting kvVersion=2 will take this into account.

With your config server running, you can make HTTP requests to the server to retrieve values from the Vault backend. To do so, you need a token for your Vault server.

First, place some data in you Vault, as shown in the following example:

$ vault write secret/application foo=bar baz=bam

$ vault write secret/myapp foo=myappsbar

Second, make an HTTP request to your config server to retrieve the values, as shown in the following example:

$ curl -X "GET" "http://localhost:8888/myapp/default" -H "X-Config-Token: yourtoken"

You should see a response similar to the following:

**{**

"name":"myapp"**,**

"profiles":**[**

"default"

]**,**

"label":null**,**

"version":null**,**

"state":null**,**

"propertySources":**[**

**{**

"name":"vault:myapp"**,**

"source":**{**

"foo":"myappsbar"

**}**

**},**

**{**

"name":"vault:application"**,**

"source":**{**

"baz":"bam"**,**

"foo":"bar"

**}**

**}**

**]**

**}**

**Multiple Properties Sources**

When using Vault, you can provide your applications with multiple properties sources. For example, assume you have written data to the following paths in Vault:

secret/myApp,dev

secret/myApp

secret/application,dev

secret/application

Properties written to secret/application are available to [all applications using the Config Server](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html). An application with the name, myApp, would have any properties written to secret/myApp and secret/application available to it. When myApp has the dev profile enabled, properties written to all of the above paths would be available to it, with properties in the first path in the list taking priority over the others.

**5.1.5 Accessing Backends Through a Proxy**

The configuration server can access a Git or Vault backend through an HTTP or HTTPS proxy. This behavior is controlled for either Git or Vault by settings under proxy.http and proxy.https. These settings are per repository, so if you are using a [composite environment repository](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#composite-environment-repositories) you must configure proxy settings for each backend in the composite individually. If using a network which requires separate proxy servers for HTTP and HTTPS URLs, you can configure both the HTTP and the HTTPS proxy settings for a single backend.

The following table describes the proxy configuration properties for both HTTP and HTTPS proxies. All of these properties must be prefixed by proxy.http or proxy.https.

**Table 5.2. Proxy Configuration Properties**

| **Property Name** | **Remarks** |
| --- | --- |
| **host** | The host of the proxy. |
| **port** | The port with which to access the proxy. |
| **nonProxyHosts** | Any hosts which the configuration server should access outside the proxy. If values are provided for both proxy.http.nonProxyHosts and proxy.https.nonProxyHosts, the proxy.http value will be used. |
| **username** | The username with which to authenticate to the proxy. If values are provided for both proxy.http.username and proxy.https.username, the proxy.http value will be used. |
| **password** | The password with which to authenticate to the proxy. If values are provided for both proxy.http.password and proxy.https.password, the proxy.http value will be used. |

The following configuration uses an HTTPS proxy to access a Git repository.

spring:

profiles:

active: git

cloud:

config:

server:

git:

uri: https://github.com/spring-cloud-samples/config-repo

proxy:

https:

host: my-proxy.host.io

password: myproxypassword

port: '3128'

username: myproxyusername

nonProxyHosts: example.com

**5.1.6 Sharing Configuration With All Applications**

Sharing configuration between all applications varies according to which approach you take, as described in the following topics:

* [the section called “File Based Repositories”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-config-server-file-based-repositories)
* [the section called “Vault Server”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-config-server-vault-server)

**File Based Repositories**

With file-based (git, svn, and native) repositories, resources with file names in application\* (application.properties, application.yml, application-\*.properties, and so on) are shared between all client applications. You can use resources with these file names to configure global defaults and have them be overridden by application-specific files as necessary.

The #\_property\_overrides[property overrides] feature can also be used for setting global defaults, with placeholders applications allowed to override them locally.

|  |
| --- |
| [Tip] |
| With the “native” profile (a local file system backend) , you should use an explicit search location that is not part of the server’s own configuration. Otherwise, the application\* resources in the default search locations get removed because they are part of the server. |

**Vault Server**

When using Vault as a backend, you can share configuration with all applications by placing configuration in secret/application. For example, if you run the following Vault command, all applications using the config server will have the properties foo and baz available to them:

$ vault write secret/application foo=bar baz=bam

**5.1.7 JDBC Backend**

Spring Cloud Config Server supports JDBC (relational database) as a backend for configuration properties. You can enable this feature by adding spring-jdbc to the classpath and using the jdbc profile or by adding a bean of type JdbcEnvironmentRepository. If you include the right dependencies on the classpath (see the user guide for more details on that), Spring Boot configures a data source.

The database needs to have a table called PROPERTIES with columns called APPLICATION, PROFILE, and LABEL (with the usual Environment meaning), plus KEYand VALUE for the key and value pairs in Properties style. All fields are of type String in Java, so you can make them VARCHAR of whatever length you need. Property values behave in the same way as they would if they came from Spring Boot properties files named {application}-{profile}.properties, including all the encryption and decryption, which will be applied as post-processing steps (that is, not in the repository implementation directly).

**5.1.8 Composite Environment Repositories**

In some scenarios, you may wish to pull configuration data from multiple environment repositories. To do so, you can enable the composite profile in your configuration server’s application properties or YAML file. If, for example, you want to pull configuration data from a Subversion repository as well as two Git repositories, you can set the following properties for your configuration server:

spring:

profiles:

active: composite

cloud:

config:

server:

composite:

-

type: svn

uri: file:///path/to/svn/repo

-

type: git

uri: file:///path/to/rex/git/repo

-

type: git

uri: file:///path/to/walter/git/repo

Using this configuration, precedence is determined by the order in which repositories are listed under the composite key. In the above example, the Subversion repository is listed first, so a value found in the Subversion repository will override values found for the same property in one of the Git repositories. A value found in the rex Git repository will be used before a value found for the same property in the walter Git repository.

If you want to pull configuration data only from repositories that are each of distinct types, you can enable the corresponding profiles, rather than the composite profile, in your configuration server’s application properties or YAML file. If, for example, you want to pull configuration data from a single Git repository and a single HashiCorp Vault server, you can set the following properties for your configuration server:

spring:

profiles:

active: git**,** vault

cloud:

config:

server:

git:

uri: file:///path/to/git/repo

order: 2

vault:

host: 127.0.0.1

port: 8200

order: 1

Using this configuration, precedence can be determined by an order property. You can use the order property to specify the priority order for all your repositories. The lower the numerical value of the order property, the higher priority it has. The priority order of a repository helps resolve any potential conflicts between repositories that contain values for the same properties.

|  |
| --- |
| [Note] |
| If your composite environment includes a Vault server as in the previous example, you must include a Vault token in every request made to the configuration server. See [Vault Backend](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault-backend). |
| [Note] |
| Any type of failure when retrieving values from an environment repository results in a failure for the entire composite environment. | |

|  |
| --- |
| [Note] |
| When using a composite environment, it is important that all repositories contain the same labels. If you have an environment similar to those in the preceding examples and you request configuration data with the master label but the Subversion repository does not contain a branch called master, the entire request fails. |

**Custom Composite Environment Repositories**

In addition to using one of the environment repositories from Spring Cloud, you can also provide your own EnvironmentRepository bean to be included as part of a composite environment. To do so, your bean must implement the EnvironmentRepository interface. If you want to control the priority of your custom EnvironmentRepository within the composite environment, you should also implement the Ordered interface and override the getOrdered method. If you do not implement the Ordered interface, your EnvironmentRepository is given the lowest priority.

**5.1.9 Property Overrides**

The Config Server has an “overrides” feature that lets the operator provide configuration properties to all applications. The overridden properties cannot be accidentally changed by the application with the normal Spring Boot hooks. To declare overrides, add a map of name-value pairs to spring.cloud.config.server.overrides, as shown in the following example:

spring:

cloud:

config:

server:

overrides:

foo: bar

The preceding examples causes all applications that are config clients to read foo=bar, independent of their own configuration.

|  |
| --- |
| [Note] |
| A configuration system cannot force an application to use configuration data in any particular way. Consequently, overrides are not enforceable. However, they do provide useful default behavior for Spring Cloud Config clients. |
| [Tip] |
| Normally, Spring environment placeholders with ${} can be escaped (and resolved on the client) by using backslash (\) to escape the $ or the {. For example, \${app.foo:bar} resolves to bar, unless the app provides its own app.foo. | |

|  |
| --- |
| [Note] |
| In YAML, you do not need to escape the backslash itself. However, in properties files, you do need to escape the backslash, when you configure the overrides on the server. |

You can change the priority of all overrides in the client to be more like default values, letting applications supply their own values in environment variables or System properties, by setting the spring.cloud.config.overrideNone=true flag (the default is false) in the remote repository.

**5.2 Health Indicator**

Config Server comes with a Health Indicator that checks whether the configured EnvironmentRepository is working. By default, it asks the EnvironmentRepositoryfor an application named app, the default profile, and the default label provided by the EnvironmentRepository implementation.

You can configure the Health Indicator to check more applications along with custom profiles and custom labels, as shown in the following example:

spring:

cloud:

config:

server:

health:

repositories:

myservice:

label: mylabel

myservice-dev:

name: myservice

profiles: development

You can disable the Health Indicator by setting spring.cloud.config.server.health.enabled=false.

**5.3 Security**

You can secure your Config Server in any way that makes sense to you (from physical network security to OAuth2 bearer tokens), because Spring Security and Spring Boot offer support for many security arrangements.

To use the default Spring Boot-configured HTTP Basic security, include Spring Security on the classpath (for example, through spring-boot-starter-security). The default is a username of user and a randomly generated password. A random password is not useful in practice, so we recommend you configure the password (by setting spring.security.user.password) and encrypt it (see below for instructions on how to do that).

**5.4 Encryption and Decryption**

|  |  |
| --- | --- |
| [Important] | **Important** |
| To use the encryption and decryption features you need the full-strength JCE installed in your JVM (it is not included by default). You can download the “Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files” from Oracle and follow the installation instructions (essentially, you need to replace the two policy files in the JRE lib/security directory with the ones that you downloaded). |

If the remote property sources contain encrypted content (values starting with {cipher}), they are decrypted before sending to clients over HTTP. The main advantage of this setup is that the property values need not be in plain text when they are “at rest” (for example, in a git repository). If a value cannot be decrypted, it is removed from the property source and an additional property is added with the same key but prefixed with invalid and a value that means “not applicable” (usually <n/a>). This is largely to prevent cipher text being used as a password and accidentally leaking.

If you set up a remote config repository for config client applications, it might contain an application.yml similar to the following:

**application.yml.**

spring:

datasource:

username: dbuser

password: '{cipher}FKSAJDFGYOS8F7GLHAKERGFHLSAJ'

Encrypted values in a .properties file must not be wrapped in quotes. Otherwise, the value is not decrypted. The following example shows values that would work:

**application.properties.**

spring.datasource.username: dbuser

spring.datasource.password: {cipher}FKSAJDFGYOS8F7GLHAKERGFHLSAJ

You can safely push this plain text to a shared git repository, and the secret password remains protected.

The server also exposes /encrypt and /decrypt endpoints (on the assumption that these are secured and only accessed by authorized agents). If you edit a remote config file, you can use the Config Server to encrypt values by POSTing to the /encrypt endpoint, as shown in the following example:

$ curl localhost:8888/encrypt -d mysecret

682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda

|  |
| --- |
| [Note] |
| If the value you encrypt has characters in it that need to be URL encoded, you should use the --data-urlencode option to curl to make sure they are encoded properly. |
| [Tip] |
| Be sure not to include any of the curl command statistics in the encrypted value. Outputting the value to a file can help avoid this problem. | |

The inverse operation is also available through /decrypt (provided the server is configured with a symmetric key or a full key pair), as shown in the following example:

$ curl localhost:8888/decrypt -d 682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda

mysecret

|  |
| --- |
| [Tip] |
| If you testing with curl, then use --data-urlencode (instead of -d) or set an explicit Content-Type: text/plain to make sure curl encodes the data correctly when there are special characters ('+' is particularly tricky). |

Take the encrypted value and add the {cipher} prefix before you put it in the YAML or properties file and before you commit and push it to a remote (potentially insecure) store.

The /encrypt and /decrypt endpoints also both accept paths in the form of /\*/{name}/{profiles}, which can be used to control cryptography on a per-application (name) and per-profile basis when clients call into the main environment resource.

|  |
| --- |
| [Note] |
| To control the cryptography in this granular way, you must also provide a @Bean of type TextEncryptorLocator that creates a different encryptor per name and profiles. The one that is provided by default does not do so (all encryptions use the same key). |

The spring command line client (with Spring Cloud CLI extensions installed) can also be used to encrypt and decrypt, as shown in the following example:

$ spring encrypt mysecret --key foo

682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda

$ spring decrypt --key foo 682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda

mysecret

To use a key in a file (such as an RSA public key for encryption), prepend the key value with "@" and provide the file path, as shown in the following example:

$ spring encrypt mysecret --key @${HOME}/.ssh/id\_rsa.pub

AQAjPgt3eFZQXwt8tsHAVv/QHiY5sI2dRcR+...

|  |
| --- |
| [Note] |
| The --key argument is mandatory (despite having a -- prefix). |

**5.5 Key Management**

The Config Server can use a symmetric (shared) key or an asymmetric one (RSA key pair). The asymmetric choice is superior in terms of security, but it is often more convenient to use a symmetric key since it is a single property value to configure in the bootstrap.properties.

To configure a symmetric key, you need to set encrypt.key to a secret String (or use the ENCRYPT\_KEY environment variable to keep it out of plain-text configuration files).

To configure an asymmetric key, you can either set the key as a PEM-encoded text value (in encrypt.key) or use a keystore (such as the keystore created by the keytool utility that comes with the JDK). The following table describes the keystore properties:

| **Property** | **Description** |
| --- | --- |
| encrypt.keyStore.location | Contains a Resource location |
| encrypt.keyStore.password | Holds the password that unlocks the keystore |
| encrypt.keyStore.alias | Identifies which key in the store to use |

The encryption is done with the public key, and a private key is needed for decryption. Thus, in principle, you can configure only the public key in the server if you want to only encrypt (and are prepared to decrypt the values yourself locally with the private key). In practice, you might not want to do decrypt locally, because it spreads the key management process around all the clients, instead of concentrating it in the server. On the other hand, it can be a useful option if your config server is relatively insecure and only a handful of clients need the encrypted properties.

**5.6 Creating a Key Store for Testing**

To create a keystore for testing, you can use a command resembling the following:

$ keytool -genkeypair -alias mytestkey -keyalg RSA \

-dname "CN=Web Server,OU=Unit,O=Organization,L=City,S=State,C=US" \

-keypass changeme -keystore server.jks -storepass letmein

Put the server.jks file in the classpath (for instance) and then, in your bootstrap.yml, for the Config Server, create the following settings:

encrypt:

keyStore:

location: classpath:/server.jks

password: letmein

alias: mytestkey

secret: changeme

**5.7 Using Multiple Keys and Key Rotation**

In addition to the {cipher} prefix in encrypted property values, the Config Server looks for zero or more {name:value} prefixes before the start of the (Base64 encoded) cipher text. The keys are passed to a TextEncryptorLocator, which can do whatever logic it needs to locate a TextEncryptor for the cipher. If you have configured a keystore (encrypt.keystore.location), the default locator looks for keys with aliases supplied by the key prefix, with a cipher text like resembling the following:

foo:

bar: `{cipher}{key:testkey}...`

The locator looks for a key named "testkey". A secret can also be supplied by using a {secret:…​} value in the prefix. However, if it is not supplied, the default is to use the keystore password (which is what you get when you build a keytore and do not specify a secret). If you do supply a secret, you should also encrypt the secret using a custom SecretLocator.

When the keys are being used only to encrypt a few bytes of configuration data (that is, they are not being used elsewhere), key rotation is hardly ever necessary on cryptographic grounds. However, you might occasionally need to change the keys (for example, in the event of a security breach). In that case, all the clients would need to change their source config files (for example, in git) and use a new {key:…​} prefix in all the ciphers. Note that the clients need to first check that the key alias is available in the Config Server keystore.

|  |
| --- |
| [Tip] |
| If you want to let the Config Server handle all encryption as well as decryption, the {name:value} prefixes can also be added as plain text posted to the /encrypt endpoint, . |

**5.8 Serving Encrypted Properties**

Sometimes you want the clients to decrypt the configuration locally, instead of doing it in the server. In that case, if you provide the encrypt.\* configuration to locate a key, you can still have /encrypt and /decrypt endpoints, but you need to explicitly switch off the decryption of outgoing properties by placing spring.cloud.config.server.encrypt.enabled=false in bootstrap.[yml|properties]. If you do not care about the endpoints, it should work if you do not configure either the key or the enabled flag.

**6. Serving Alternative Formats**

The default JSON format from the environment endpoints is perfect for consumption by Spring applications, because it maps directly onto the Environment abstraction. If you prefer, you can consume the same data as YAML or Java properties by adding a suffix (".yml", ".yaml" or ".properties") to the resource path. This can be useful for consumption by applications that do not care about the structure of the JSON endpoints or the extra metadata they provide (for example, an application that is not using Spring might benefit from the simplicity of this approach).

The YAML and properties representations have an additional flag (provided as a boolean query parameter called resolvePlaceholders) to signal that placeholders in the source documents (in the standard Spring ${…​} form) should be resolved in the output before rendering, where possible. This is a useful feature for consumers that do not know about the Spring placeholder conventions.

|  |
| --- |
| [Note] |
| There are limitations in using the YAML or properties formats, mainly in relation to the loss of metadata. For example, the JSON is structured as an ordered list of property sources, with names that correlate with the source. The YAML and properties forms are coalesced into a single map, even if the origin of the values has multiple sources, and the names of the original source files are lost. Also, the YAML representation is not necessarily a faithful representation of the YAML source in a backing repository either. It is constructed from a list of flat property sources, and assumptions have to be made about the form of the keys. |

**7. Serving Plain Text**

Instead of using the Environment abstraction (or one of the alternative representations of it in YAML or properties format), your applications might need generic plain-text configuration files that are tailored to their environment. The Config Server provides these through an additional endpoint at /{name}/{profile}/{label}/{path}, where name, profile, and label have the same meaning as the regular environment endpoint, but path is a file name (such as log.xml). The source files for this endpoint are located in the same way as for the environment endpoints. The same search path is used for properties and YAML files. However, instead of aggregating all matching resources, only the first one to match is returned.

After a resource is located, placeholders in the normal format (${…​}) are resolved by using the effective Environment for the supplied application name, profile, and label. In this way, the resource endpoint is tightly integrated with the environment endpoints. Consider the following example for a GIT or SVN repository:

application.yml

nginx.conf

where nginx.conf looks like this:

server {

listen 80;

server\_name ${nginx.server.name};

}

and application.yml like this:

nginx:

server:

name: example.com

*---*

spring:

profiles: development

nginx:

server:

name: develop.com

The /foo/default/master/nginx.conf resource might be as follows:

server {

listen 80;

server\_name example.com;

}

and /foo/development/master/nginx.conf like this:

server {

listen 80;

server\_name develop.com;

}

|  |
| --- |
| [Note] |
| As with the source files for environment configuration, the profile is used to resolve the file name. So, if you want a profile-specific file, /\*/development/\*/logback.xml can be resolved by a file called logback-development.xml (in preference to logback.xml). |
| [Note] |
| If you do not want to supply the label and let the server use the default label, you can supply a useDefaultLabel request parameter. So, the preceding example for the default profile could be /foo/default/nginx.conf?useDefaultLabel. | |

**8. Embedding the Config Server**

The Config Server runs best as a standalone application. However, if need be, you can embed it in another application. To do so, use the @EnableConfigServerannotation. An optional property named spring.cloud.config.server.bootstrap can be useful in this case is. It is a flag to indicate whether the server should configure itself from its own remote repository. By default, the flag is off, because it can delay startup. However, when embedded in another application, it makes sense to initialize the same way as any other application.

|  |
| --- |
| [Note] |
| If you use the bootstrap flag, the config server needs to have its name and repository URI configured in bootstrap.yml. |

To change the location of the server endpoints, you can (optionally) set spring.cloud.config.server.prefix (for example, /config), to serve the resources under a prefix. The prefix should start but not end with a /. It is applied to the @RequestMappings in the Config Server (that is, underneath the Spring Boot server.servletPath and server.contextPath prefixes).

If you want to read the configuration for an application directly from the backend repository (instead of from the config server), you basically wat an embedded config server with no endpoints. You can switch off the endpoints entirely by not using the @EnableConfigServer annotation (set spring.cloud.config.server.bootstrap=true).

**9. Push Notifications and Spring Cloud Bus**

Many source code repository providers (such as Github, Gitlab, Gitee, or Bitbucket) notify you of changes in a repository through a webhook. You can configure the webhook through the provider’s user interface as a URL and a set of events in which you are interested. For instance, [Github](https://developer.github.com/v3/activity/events/types/#pushevent) uses a POST to the webhook with a JSON body containing a list of commits and a header (X-Github-Event) set to push. If you add a dependency on the spring-cloud-config-monitor library and activate the Spring Cloud Bus in your Config Server, then a /monitor endpoint is enabled.

When the webhook is activated, the Config Server sends a RefreshRemoteApplicationEvent targeted at the applications it thinks might have changed. The change detection can be strategized. However, by default, it looks for changes in files that match the application name (for example, foo.properties is targeted at the fooapplication, while application.properties is targeted at all applications). The strategy to use when you want to override the behavior is PropertyPathNotificationExtractor, which accepts the request headers and body as parameters and returns a list of file paths that changed.

The default configuration works out of the box with Github, Gitlab, Gitee, or Bitbucket. In addition to the JSON notifications from Github, Gitlab, Gitee, or Bitbucket, you can trigger a change notification by POSTing to /monitor with form-encoded body parameters in the pattern of path={name}. Doing so broadcasts to applications matching the {name} pattern (which can contain wildcards).

|  |
| --- |
| [Note] |
| The RefreshRemoteApplicationEvent is transmitted only if the spring-cloud-bus is activated in both the Config Server and in the client application. |
| [Note] |
| The default configuration also detects filesystem changes in local git repositories. In that case, the webhook is not used. However, as soon as you edit a config file, a refresh is broadcast. | |

**10. Spring Cloud Config Client**

A Spring Boot application can take immediate advantage of the Spring Config Server (or other external property sources provided by the application developer). It also picks up some additional useful features related to Environment change events.

**10.1 Config First Bootstrap**

The default behavior for any application that has the Spring Cloud Config Client on the classpath is as follows: When a config client starts, it binds to the Config Server (through the spring.cloud.config.uri bootstrap configuration property) and initializes Spring Environment with remote property sources.

The net result of this behavior is that all client applciations that want to consume the Config Server need a bootstrap.yml (or an environment variable) with the server address set in spring.cloud.config.uri (it defaults to "http://localhost:8888").

**10.2 Discovery First Bootstrap**

If you use a `DiscoveryClient implementation, such as Spring Cloud Netflix and Eureka Service Discovery or Spring Cloud Consul, you can have the Config Server register with the Discovery Service. However, in the default “Config First” mode, clients cannot take advantage of the registration.

If you prefer to use DiscoveryClient to locate the Config Server, you can do so by setting spring.cloud.config.discovery.enabled=true (the default is false). The net result of doing so is that client applications all need a bootstrap.yml (or an environment variable) with the appropriate discovery configuration. For example, with Spring Cloud Netflix, you need to define the Eureka server address (for example, in eureka.client.serviceUrl.defaultZone). The price for using this option is an extra network round trip on startup, to locate the service registration. The benefit is that, as long as the Discovery Service is a fixed point, the Config Server can change its coordinates. The default service ID is configserver, but you can change that on the client by setting spring.cloud.config.discovery.serviceId(and on the server, in the usual way for a service, such as by setting spring.application.name).

The discovery client implementations all support some kind of metadata map (for example, we have eureka.instance.metadataMap for Eureka). Some additional properties of the Config Server may need to be configured in its service registration metadata so that clients can connect correctly. If the Config Server is secured with HTTP Basic, you can configure the credentials as username and password. Also, if the Config Server has a context path, you can set configPath. For example, the following YAML file is for a Config Server that is a Eureka client:

**bootstrap.yml.**

eureka:

instance:

...

metadataMap:

user: osufhalskjrtl

password: lviuhlszvaorhvlo5847

configPath: /config

**10.3 Config Client Fail Fast**

In some cases, you may want to fail startup of a service if it cannot connect to the Config Server. If this is the desired behavior, set the bootstrap configuration property spring.cloud.config.fail-fast=true to make the client halt with an Exception.

**10.4 Config Client Retry**

If you expect that the config server may occasionally be unavailable when your application starts, you can make it keep trying after a failure. First, you need to set spring.cloud.config.fail-fast=true. Then you need to add spring-retry and spring-boot-starter-aop to your classpath. The default behavior is to retry six times with an initial backoff interval of 1000ms and an exponential multiplier of 1.1 for subsequent backoffs. You can configure these properties (and others) by setting the spring.cloud.config.retry.\* configuration properties.

|  |
| --- |
| [Tip] |
| To take full control of the retry behavior, add a @Bean of type RetryOperationsInterceptor with an ID of configServerRetryInterceptor. Spring Retry has a RetryInterceptorBuilder that supports creating one. |

**10.5 Locating Remote Configuration Resources**

The Config Service serves property sources from /{name}/{profile}/{label}, where the default bindings in the client app are as follows:

* "name" = ${spring.application.name}
* "profile" = ${spring.profiles.active} (actually Environment.getActiveProfiles())
* "label" = "master"

You can override all of them by setting spring.cloud.config.\* (where \* is name, profile or label). The label is useful for rolling back to previous versions of configuration. With the default Config Server implementation, it can be a git label, branch name, or commit ID. Label can also be provided as a comma-separated list. In that case, the items in the list are tried one by one until one succeeds. This behavior can be useful when working on a feature branch. For instance, you might want to align the config label with your branch but make it optional (in that case, use spring.cloud.config.label=myfeature,develop).

**10.6 Specifying Multiple Urls for the Config Server**

To ensure high availability when you have multiple instances of Config Server deployed and expect one or more instances to be unavailable from time to time, you can either specify multiple URLs (as a comma-separated list under the spring.cloud.config.uri property) or have all your instances register in a Service Registry like Eureka ( if using Discovery-First Bootstrap mode ). Note that doing so ensures high availability only when the Config Server is not running (that is, when the application has exited) or when a connection timeout has occurred. For example, if the Config Server returns a 500 (Internal Server Error) response or the Config Client receives a 401 from the Config Server (due to bad credentials or other causes), the Config Client does not try to fetch properties from other URLs. An error of that kind indicates a user issue rather than an availability problem.

If you use HTTP basic security on your Config Server, it is currently possible to support per-Config Server auth credentials only if you embed the credentials in each URL you specify under the spring.cloud.config.uri property. If you use any other kind of security mechanism, you cannot (currently) support per-Config Server authentication and authorization.

**10.7 Configuring Read Timeouts**

If you want to configure read timeout, this can be done by using the property spring.cloud.config.request-read-timeout.

**10.8 Security**

If you use HTTP Basic security on the server, clients need to know the password (and username if it is not the default). You can specify the username and password through the config server URI or via separate username and password properties, as shown in the following example:

**bootstrap.yml.**

spring:

cloud:

config:

uri: https://user:secret@myconfig.mycompany.com

The following example shows an alternate way to pass the same information:

**bootstrap.yml.**

spring:

cloud:

config:

uri: https://myconfig.mycompany.com

username: user

password: secret

The spring.cloud.config.password and spring.cloud.config.username values override anything that is provided in the URI.

If you deploy your apps on Cloud Foundry, the best way to provide the password is through service credentials (such as in the URI, since it does not need to be in a config file). The following example works locally and for a user-provided service on Cloud Foundry named configserver:

**bootstrap.yml.**

spring:

cloud:

config:

uri: ${vcap.services.configserver.credentials.uri:http://user:password@localhost:8888**}**

If you use another form of security, you might need to [provide a RestTemplate](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#custom-rest-template) to the ConfigServicePropertySourceLocator (for example, by grabbing it in the bootstrap context and injecting it).

**10.8.1 Health Indicator**

The Config Client supplies a Spring Boot Health Indicator that attempts to load configuration from the Config Server. The health indicator can be disabled by setting health.config.enabled=false. The response is also cached for performance reasons. The default cache time to live is 5 minutes. To change that value, set the health.config.time-to-live property (in milliseconds).

**10.8.2 Providing A Custom RestTemplate**

In some cases, you might need to customize the requests made to the config server from the client. Typically, doing so involves passing special Authorizationheaders to authenticate requests to the server. To provide a custom RestTemplate:

1. Create a new configuration bean with an implementation of PropertySourceLocator, as shown in the following example:

**CustomConfigServiceBootstrapConfiguration.java.**

*@Configuration*

**public** **class** CustomConfigServiceBootstrapConfiguration {

*@Bean*

**public** ConfigServicePropertySourceLocator configServicePropertySourceLocator() {

ConfigClientProperties clientProperties = configClientProperties();

ConfigServicePropertySourceLocator configServicePropertySourceLocator = **new** ConfigServicePropertySourceLocator(clientProperties);

configServicePropertySourceLocator.setRestTemplate(customRestTemplate(clientProperties));

**return** configServicePropertySourceLocator;

}

}

1. In resources/META-INF, create a file called spring.factories and specify your custom configuration, as shown in the following example:

**spring.factories.**

org.springframework.cloud.bootstrap.BootstrapConfiguration = com.my.config.client.CustomConfigServiceBootstrapConfiguration

**10.8.3 Vault**

When using Vault as a backend to your config server, the client needs to supply a token for the server to retrieve values from Vault. This token can be provided within the client by setting spring.cloud.config.token in bootstrap.yml, as shown in the following example:

**bootstrap.yml.**

spring:

cloud:

config:

token: YourVaultToken

**10.9 Nested Keys In Vault**

Vault supports the ability to nest keys in a value stored in Vault, as shown in the following example:

echo -n '{"appA": {"secret": "appAsecret"}, "bar": "baz"}' | vault write secret/myapp -

This command writes a JSON object to your Vault. To access these values in Spring, you would use the traditional dot(.) annotation, as shown in the following example

*@Value("${appA.secret}")*

String name = "World";

The preceding code would sets the value of the name variable to appAsecret.

**Part III. Spring Cloud Netflix**

**Finchley.SR1**

This project provides Netflix OSS integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms. With a few simple annotations you can quickly enable and configure the common patterns inside your application and build large distributed systems with battle-tested Netflix components. The patterns provided include Service Discovery (Eureka), Circuit Breaker (Hystrix), Intelligent Routing (Zuul) and Client Side Load Balancing (Ribbon).

**11. Service Discovery: Eureka Clients**

Service Discovery is one of the key tenets of a microservice-based architecture. Trying to hand-configure each client or some form of convention can be difficult to do and can be brittle. Eureka is the Netflix Service Discovery Server and Client. The server can be configured and deployed to be highly available, with each server replicating state about the registered services to the others.

**11.1 How to Include Eureka Client**

To include the Eureka Client in your project, use the starter with a group ID of org.springframework.cloud and an artifact ID of spring-cloud-starter-netflix-eureka-client. See the [Spring Cloud Project page](https://projects.spring.io/spring-cloud/) for details on setting up your build system with the current Spring Cloud Release Train.

**11.2 Registering with Eureka**

When a client registers with Eureka, it provides meta-data about itself — such as host, port, health indicator URL, home page, and other details. Eureka receives heartbeat messages from each instance belonging to a service. If the heartbeat fails over a configurable timetable, the instance is normally removed from the registry.

The following example shows a minimal Eureka client application:

*@SpringBootApplication*

*@RestController*

**public** **class** Application {

*@RequestMapping("/")*

**public** String home() {

**return** "Hello world";

}

**public** **static** **void** main(String[] args) {

**new** SpringApplicationBuilder(Application.**class**).web(true).run(args);

}

}

Note that the preceding example shows a normal [Spring Boot](https://projects.spring.io/spring-boot/) application. By having spring-cloud-starter-netflix-eureka-client on the classpath, your application automatically registers with the Eureka Server. Configuration is required to locate the Eureka server, as shown in the following example:

**application.yml.**

eureka:

client:

serviceUrl:

defaultZone: http://localhost:8761/eureka/

In the preceding example, "defaultZone" is a magic string fallback value that provides the service URL for any client that does not express a preference (in other words, it is a useful default).

The default application name (that is, the service ID), virtual host, and non-secure port (taken from the Environment) are ${spring.application.name}, ${spring.application.name} and ${server.port}, respectively.

Having spring-cloud-starter-netflix-eureka-client on the classpath makes the app into both a Eureka “instance” (that is, it registers itself) and a “client” (it can query the registry to locate other services). The instance behaviour is driven by eureka.instance.\* configuration keys, but the defaults are fine if you ensure that your application has a value for spring.application.name (this is the default for the Eureka service ID or VIP).

See [EurekaInstanceConfigBean](https://github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-cloud-netflix-eureka-client/src/main/java/org/springframework/cloud/netflix/eureka/EurekaInstanceConfigBean.java) and [EurekaClientConfigBean](https://github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-cloud-netflix-eureka-client/src/main/java/org/springframework/cloud/netflix/eureka/EurekaClientConfigBean.java) for more details on the configurable options.

To disable the Eureka Discovery Client, you can set eureka.client.enabled to false.

**11.3 Authenticating with the Eureka Server**

HTTP basic authentication is automatically added to your eureka client if one of the eureka.client.serviceUrl.defaultZone URLs has credentials embedded in it (curl style, as follows: <http://user:password@localhost:8761/eureka>). For more complex needs, you can create a @Bean of type DiscoveryClientOptionalArgs and inject ClientFilter instances into it, all of which is applied to the calls from the client to the server.

|  |
| --- |
| [Note] |
| Because of a limitation in Eureka, it is not possible to support per-server basic auth credentials, so only the first set that are found is used. |

**11.4 Status Page and Health Indicator**

The status page and health indicators for a Eureka instance default to /info and /health respectively, which are the default locations of useful endpoints in a Spring Boot Actuator application. You need to change these, even for an Actuator application if you use a non-default context path or servlet path (such as server.servletPath=/custom). The following example shows the default values for the two settings:

**application.yml.**

eureka:

instance:

statusPageUrlPath: ${server.servletPath}/info

healthCheckUrlPath: ${server.servletPath}/health

These links show up in the metadata that is consumed by clients and are used in some scenarios to decide whether to send requests to your application, so it is helpful if they are accurate.

|  |
| --- |
| [Note] |
| In Dalston it was also required to set the status and health check URLs when changing that management context path. This requirement was removed beginning in Edgware. |

**11.5 Registering a Secure Application**

If your app wants to be contacted over HTTPS, you can set two flags in the EurekaInstanceConfig:

* eureka.instance.[nonSecurePortEnabled]=[false]
* eureka.instance.[securePortEnabled]=[true]

Doing so makes Eureka publish instance information that shows an explicit preference for secure communication. The Spring Cloud DiscoveryClient always returns a URI starting with https for a service configured this way. Similarly, when a service is configured this way, the Eureka (native) instance information has a secure health check URL.

Because of the way Eureka works internally, it still publishes a non-secure URL for the status and home pages unless you also override those explicitly. You can use placeholders to configure the eureka instance URLs, as shown in the following example:

**application.yml.**

eureka:

instance:

statusPageUrl: https://${eureka.hostname}/info

healthCheckUrl: https://${eureka.hostname}/health

homePageUrl: https://${eureka.hostname}/

(Note that ${eureka.hostname} is a native placeholder only available in later versions of Eureka. You could achieve the same thing with Spring placeholders as well — for example, by using ${eureka.instance.hostName}.)

|  |
| --- |
| [Note] |
| If your application runs behind a proxy, and the SSL termination is in the proxy (for example, if you run in Cloud Foundry or other platforms as a service), then you need to ensure that the proxy “forwarded” headers are intercepted and handled by the application. If the Tomcat container embedded in a Spring Boot application has explicit configuration for the 'X-Forwarded-\\*` headers, this happens automatically. The links rendered by your app to itself being wrong (the wrong host, port, or protocol) is a sign that you got this configuration wrong. |

**11.6 Eureka’s Health Checks**

By default, Eureka uses the client heartbeat to determine if a client is up. Unless specified otherwise, the Discovery Client does not propagate the current health check status of the application, per the Spring Boot Actuator. Consequently, after successful registration, Eureka always announces that the application is in 'UP' state. This behavior can be altered by enabling Eureka health checks, which results in propagating application status to Eureka. As a consequence, every other application does not send traffic to applications in states other then 'UP'. The following example shows how to enable health checks for the client:

**application.yml.**

eureka:

client:

healthcheck:

enabled: true

|  |
| --- |
| [Warning] |
| eureka.client.healthcheck.enabled=true should only be set in application.yml. Setting the value in bootstrap.yml causes undesirable side effects, such as registering in Eureka with an UNKNOWN status. |

If you require more control over the health checks, consider implementing your own com.netflix.appinfo.HealthCheckHandler.

**11.7 Eureka Metadata for Instances and Clients**

It is worth spending a bit of time understanding how the Eureka metadata works, so you can use it in a way that makes sense in your platform. There is standard metadata for information such as hostname, IP address, port numbers, the status page, and health check. These are published in the service registry and used by clients to contact the services in a straightforward way. Additional metadata can be added to the instance registration in the eureka.instance.metadataMap, and this metadata is accessible in the remote clients. In general, additional metadata does not change the behavior of the client, unless the client is made aware of the meaning of the metadata. There are a couple of special cases, described later in this document, where Spring Cloud already assigns meaning to the metadata map.

**11.7.1 Using Eureka on Cloud Foundry**

Cloud Foundry has a global router so that all instances of the same app have the same hostname (other PaaS solutions with a similar architecture have the same arrangement). This is not necessarily a barrier to using Eureka. However, if you use the router (recommended or even mandatory, depending on the way your platform was set up), you need to explicitly set the hostname and port numbers (secure or non-secure) so that they use the router. You might also want to use instance metadata so that you can distinguish between the instances on the client (for example, in a custom load balancer). By default, the eureka.instance.instanceId is vcap.application.instance\_id, as shown in the following example:

**application.yml.**

eureka:

instance:

hostname: ${vcap.application.uris[0]}

nonSecurePort: 80

Depending on the way the security rules are set up in your Cloud Foundry instance, you might be able to register and use the IP address of the host VM for direct service-to-service calls. This feature is not yet available on Pivotal Web Services ([PWS](https://run.pivotal.io/)).

**11.7.2 Using Eureka on AWS**

If the application is planned to be deployed to an AWS cloud, the Eureka instance must be configured to be AWS-aware. You can do so by customizing the [EurekaInstanceConfigBean](https://github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-cloud-netflix-eureka-client/src/main/java/org/springframework/cloud/netflix/eureka/EurekaInstanceConfigBean.java) as follows:

*@Bean*

*@Profile("!default")*

**public** EurekaInstanceConfigBean eurekaInstanceConfig(InetUtils inetUtils) {

EurekaInstanceConfigBean b = **new** EurekaInstanceConfigBean(inetUtils);

AmazonInfo info = AmazonInfo.Builder.newBuilder().autoBuild("eureka");

b.setDataCenterInfo(info);

**return** b;

}

**11.7.3 Changing the Eureka Instance ID**

A vanilla Netflix Eureka instance is registered with an ID that is equal to its host name (that is, there is only one service per host). Spring Cloud Eureka provides a sensible default, which is defined as follows:

${spring.cloud.client.hostname}:${spring.application.name}:${spring.application.instance\_id:${server.port}}}

An example is myhost:myappname:8080.

By using Spring Cloud, you can override this value by providing a unique identifier in eureka.instance.instanceId, as shown in the following example:

**application.yml.**

eureka:

instance:

instanceId: ${spring.application.name}:${vcap.application.instance\_id:${spring.application.instance\_id:${random.value}}}

With the metadata shown in the preceding example and multiple service instances deployed on localhost, the random value is inserted there to make the instance unique. In Cloud Foundry, the vcap.application.instance\_id is populated automatically in a Spring Boot application, so the random value is not needed.

**11.8 Using the EurekaClient**

Once you have an application that is a discovery client, you can use it to discover service instances from the [Eureka Server](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-eureka-server). One way to do so is to use the native com.netflix.discovery.EurekaClient (as opposed to the Spring Cloud DiscoveryClient), as shown in the following example:

@Autowired

private EurekaClient discoveryClient;

public String serviceUrl() {

InstanceInfo instance = discoveryClient.getNextServerFromEureka("STORES", false);

return instance.getHomePageUrl();

}

|  |
| --- |
| [Tip] |
| Do not use the EurekaClient in a @PostConstruct method or in a @Scheduled method (or anywhere where the ApplicationContext might not be started yet). It is initialized in a SmartLifecycle (with phase=0), so the earliest you can rely on it being available is in another SmartLifecycle with a higher phase. |

**11.8.1 EurekaClient without Jersey**

By default, EurekaClient uses Jersey for HTTP communication. If you wish to avoid dependencies from Jersey, you can exclude it from your dependencies. Spring Cloud auto-configures a transport client based on Spring RestTemplate. The following example shows Jersey being excluded:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

<exclusions>

<exclusion>

<groupId>com.sun.jersey</groupId>

<artifactId>jersey-client</artifactId>

</exclusion>

<exclusion>

<groupId>com.sun.jersey</groupId>

<artifactId>jersey-core</artifactId>

</exclusion>

<exclusion>

<groupId>com.sun.jersey.contribs</groupId>

<artifactId>jersey-apache-client4</artifactId>

</exclusion>

</exclusions>

</dependency>

**11.9 Alternatives to the Native Netflix EurekaClient**

You need not use the raw Netflix EurekaClient. Also, it is usually more convenient to use it behind a wrapper of some sort. Spring Cloud has support for [Feign](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-feign) (a REST client builder) and [Spring RestTemplate](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-ribbon) through the logical Eureka service identifiers (VIPs) instead of physical URLs. To configure Ribbon with a fixed list of physical servers, you can set <client>.ribbon.listOfServers to a comma-separated list of physical addresses (or hostnames), where <client> is the ID of the client.

You can also use the org.springframework.cloud.client.discovery.DiscoveryClient, which provides a simple API (not specific to Netflix) for discovery clients, as shown in the following example:

@Autowired

private DiscoveryClient discoveryClient;

public String serviceUrl() {

List<ServiceInstance> list = discoveryClient.getInstances("STORES");

if (list != null && list.size() > 0 ) {

return list.get(0).getUri();

}

return null;

}

**11.10 Why Is It so Slow to Register a Service?**

Being an instance also involves a periodic heartbeat to the registry (through the client’s serviceUrl) with a default duration of 30 seconds. A service is not available for discovery by clients until the instance, the server, and the client all have the same metadata in their local cache (so it could take 3 heartbeats). You can change the period by setting eureka.instance.leaseRenewalIntervalInSeconds. Setting it to a value of less than 30 speeds up the process of getting clients connected to other services. In production, it is probably better to stick with the default, because of internal computations in the server that make assumptions about the lease renewal period.

**11.11 Zones**

If you have deployed Eureka clients to multiple zones, you may prefer that those clients use services within the same zone before trying services in another zone. To set that up, you need to configure your Eureka clients correctly.

First, you need to make sure you have Eureka servers deployed to each zone and that they are peers of each other. See the section on [zones and regions](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-eureka-server-zones-and-regions) for more information.

Next, you need to tell Eureka which zone your service is in. You can do so by using the metadataMap property. For example, if service 1 is deployed to both zone 1and zone 2, you need to set the following Eureka properties in service 1:

**Service 1 in Zone 1**

eureka.instance.metadataMap.zone = zone1

eureka.client.preferSameZoneEureka = true

**Service 1 in Zone 2**

eureka.instance.metadataMap.zone = zone2

eureka.client.preferSameZoneEureka = true

**12. Service Discovery: Eureka Server**

This section describes how to set up a Eureka server.

**12.1 How to Include Eureka Server**

To include Eureka Server in your project, use the starter with a group ID of org.springframework.cloud and an artifact ID of spring-cloud-starter-netflix-eureka-server. See the [Spring Cloud Project page](https://projects.spring.io/spring-cloud/) for details on setting up your build system with the current Spring Cloud Release Train.

**12.2 How to Run a Eureka Server**

The following example shows a minimal Eureka server:

*@SpringBootApplication*

*@EnableEurekaServer*

**public** **class** Application {

**public** **static** **void** main(String[] args) {

**new** SpringApplicationBuilder(Application.**class**).web(true).run(args);

}

}

The server has a home page with a UI and HTTP API endpoints for the normal Eureka functionality under /eureka/\*.

The following links have some Eureka background reading: [flux capacitor](https://github.com/cfregly/fluxcapacitor/wiki/NetflixOSS-FAQ#eureka-service-discovery-load-balancer) and [google group discussion](https://groups.google.com/forum/?fromgroups#!topic/eureka_netflix/g3p2r7gHnN0).

|  |
| --- |
| [Tip] |
| Due to Gradle’s dependency resolution rules and the lack of a parent bom feature, depending on spring-cloud-starter-netflix-eureka-server can cause failures on application startup. To remedy this issue, add the Spring Boot Gradle plugin and import the Spring cloud starter parent bom as follows:  **build.gradle.**  buildscript {  dependencies {  classpath("org.springframework.boot:spring-boot-gradle-plugin:{spring-boot-docs-version}")  }  }  apply plugin: "spring-boot"  dependencyManagement {  imports {  mavenBom "org.springframework.cloud:spring-cloud-dependencies:{spring-cloud-version}"  }  } |

**12.3 High Availability, Zones and Regions**

The Eureka server does not have a back end store, but the service instances in the registry all have to send heartbeats to keep their registrations up to date (so this can be done in memory). Clients also have an in-memory cache of Eureka registrations (so they do not have to go to the registry for every request to a service).

By default, every Eureka server is also a Eureka client and requires (at least one) service URL to locate a peer. If you do not provide it, the service runs and works, but it fills your logs with a lot of noise about not being able to register with the peer.

See also [below for details of Ribbon support](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-ribbon) on the client side for Zones and Regions.

**12.4 Standalone Mode**

The combination of the two caches (client and server) and the heartbeats make a standalone Eureka server fairly resilient to failure, as long as there is some sort of monitor or elastic runtime (such as Cloud Foundry) keeping it alive. In standalone mode, you might prefer to switch off the client side behavior so that it does not keep trying and failing to reach its peers. The following example shows how to switch off the client-side behavior:

**application.yml (Standalone Eureka Server).**

server:

port: 8761

eureka:

instance:

hostname: localhost

client:

registerWithEureka: false

fetchRegistry: false

serviceUrl:

defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/

Notice that the serviceUrl is pointing to the same host as the local instance.

**12.5 Peer Awareness**

Eureka can be made even more resilient and available by running multiple instances and asking them to register with each other. In fact, this is the default behavior, so all you need to do to make it work is add a valid serviceUrl to a peer, as shown in the following example:

**application.yml (Two Peer Aware Eureka Servers).**

---

spring:

profiles: peer1

eureka:

instance:

hostname: peer1

client:

serviceUrl:

defaultZone: http://peer2/eureka/

---

spring:

profiles: peer2

eureka:

instance:

hostname: peer2

client:

serviceUrl:

defaultZone: http://peer1/eureka/

In the preceding example, we have a YAML file that can be used to run the same server on two hosts (peer1 and peer2) by running it in different Spring profiles. You could use this configuration to test the peer awareness on a single host (there is not much value in doing that in production) by manipulating /etc/hosts to resolve the host names. In fact, the eureka.instance.hostname is not needed if you are running on a machine that knows its own hostname (by default, it is looked up by using java.net.InetAddress).

You can add multiple peers to a system, and, as long as they are all connected to each other by at least one edge, they synchronize the registrations amongst themselves. If the peers are physically separated (inside a data center or between multiple data centers), then the system can, in principle, survive “split-brain” type failures.

**12.6 When to Prefer IP Address**

In some cases, it is preferable for Eureka to advertise the IP addresses of services rather than the hostname. Set eureka.instance.preferIpAddress to true and, when the application registers with eureka, it uses its IP address rather than its hostname.

|  |
| --- |
| [Tip] |
| If the hostname cannot be determined by Java, then the IP address is sent to Eureka. Only explict way of setting the hostname is by setting eureka.instance.hostname property. You can set your hostname at the run-time by using an environment variable — for example, eureka.instance.hostname=${HOST\_NAME}. |

**12.7 Securing The Eureka Server**

You can secure your Eureka server simply by adding Spring Security to your server’s classpath via spring-boot-starter-security. By default when Spring Security is on the classpath it will require that a valid CSRF token be sent with every request to the app. Eureka clients will not generally possess a valid cross site request forgery (CSRF) token you will need to disable this requirement for the /eureka/\*\* endpoints. For example:

*@EnableWebSecurity*

**class** WebSecurityConfig **extends** WebSecurityConfigurerAdapter {

*@Override*

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http.csrf().ignoringAntMatchers("/eureka/\*\*");

**super**.configure(http);

}

}

For more information on CSRF see the [Spring Security documentation](https://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/#csrf).

A demo Eureka Server can be found in the Spring Cloud Samples [repo](https://github.com/spring-cloud-samples/eureka/tree/Eureka-With-Security).

**13. Circuit Breaker: Hystrix Clients**

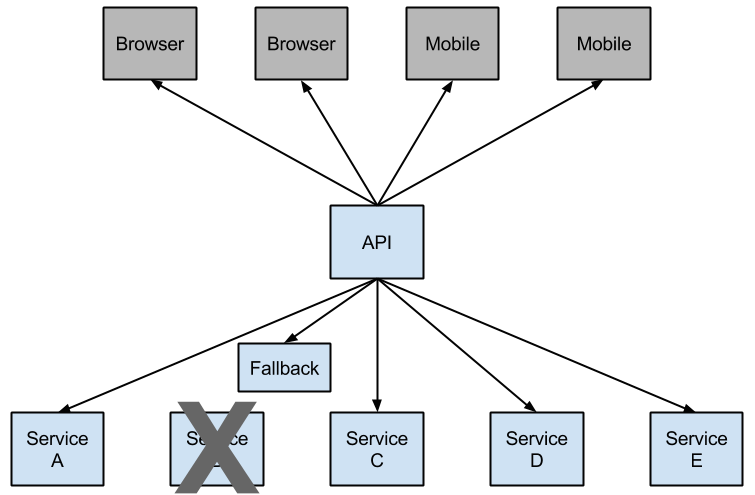
Netflix has created a library called [Hystrix](https://github.com/Netflix/Hystrix) that implements the [circuit breaker pattern](http://martinfowler.com/bliki/CircuitBreaker.html). In a microservice architecture, it is common to have multiple layers of service calls, as shown in the following example:

**Figure 13.1. Microservice Graph**



A service failure in the lower level of services can cause cascading failure all the way up to the user. When calls to a particular service exceed circuitBreaker.requestVolumeThreshold (default: 20 requests) and the failure percentage is greater than circuitBreaker.errorThresholdPercentage(default: >50%) in a rolling window defined by metrics.rollingStats.timeInMilliseconds (default: 10 seconds), the circuit opens and the call is not made. In cases of error and an open circuit, a fallback can be provided by the developer.

**Figure 13.2. Hystrix fallback prevents cascading failures**



Having an open circuit stops cascading failures and allows overwhelmed or failing services time to recover. The fallback can be another Hystrix protected call, static data, or a sensible empty value. Fallbacks may be chained so that the first fallback makes some other business call, which in turn falls back to static data.

**13.1 How to Include Hystrix**

To include Hystrix in your project, use the starter with a group ID of org.springframework.cloud and a artifact ID of spring-cloud-starter-netflix-hystrix. See the [Spring Cloud Project page](https://projects.spring.io/spring-cloud/) for details on setting up your build system with the current Spring Cloud Release Train.

The following example shows a minimal Eureka server with a Hystrix circuit breaker:

@SpringBootApplication

@EnableCircuitBreaker

public class Application {

public static void main(String[] args) {

new SpringApplicationBuilder(Application.class).web(true).run(args);

}

}

@Component

public class StoreIntegration {

@HystrixCommand(fallbackMethod = "defaultStores")

public Object getStores(Map<String, Object> parameters) {

//do stuff that might fail

}

public Object defaultStores(Map<String, Object> parameters) {

return /\* something useful \*/;

}

}

The @HystrixCommand is provided by a Netflix contrib library called [“javanica”](https://github.com/Netflix/Hystrix/tree/master/hystrix-contrib/hystrix-javanica). Spring Cloud automatically wraps Spring beans with that annotation in a proxy that is connected to the Hystrix circuit breaker. The circuit breaker calculates when to open and close the circuit and what to do in case of a failure.

To configure the @HystrixCommand you can use the commandProperties attribute with a list of @HystrixProperty annotations. See [here](https://github.com/Netflix/Hystrix/tree/master/hystrix-contrib/hystrix-javanica#configuration) for more details. See the [Hystrix wiki](https://github.com/Netflix/Hystrix/wiki/Configuration) for details on the properties available.

**13.2 Propagating the Security Context or Using Spring Scopes**

If you want some thread local context to propagate into a @HystrixCommand, the default declaration does not work, because it executes the command in a thread pool (in case of timeouts). You can switch Hystrix to use the same thread as the caller through configuration or directly in the annotation, by asking it to use a different “Isolation Strategy”. The following example demonstrates setting the thread in the annotation:

*@HystrixCommand(fallbackMethod = "stubMyService",*

*commandProperties = {*

*@HystrixProperty(name="execution.isolation.strategy", value="SEMAPHORE")*

*}*

*)*

...

The same thing applies if you are using @SessionScope or @RequestScope. If you encounter a runtime exception that says it cannot find the scoped context, you need to use the same thread.

You also have the option to set the hystrix.shareSecurityContext property to true. Doing so auto-configures a Hystrix concurrency strategy plugin hook to transfer the SecurityContext from your main thread to the one used by the Hystrix command. Hystrix does not let multiple Hystrix concurrency strategy be registered so an extension mechanism is available by declaring your own HystrixConcurrencyStrategy as a Spring bean. Spring Cloud looks for your implementation within the Spring context and wrap it inside its own plugin.

**13.3 Health Indicator**

The state of the connected circuit breakers are also exposed in the /health endpoint of the calling application, as shown in the following example:

**{**

"hystrix": **{**

"openCircuitBreakers": **[**

"StoreIntegration::getStoresByLocationLink"

]**,**

"status": "CIRCUIT\_OPEN"

**},**

"status": "UP"

**}**

**13.4 Hystrix Metrics Stream**

To enable the Hystrix metrics stream, include a dependency on spring-boot-starter-actuator and setmanagement.endpoints.web.exposure.include: hystrix.stream. Doing so exposes the /actuator/hystrix.stream as a management endpoint, as shown in the following example:

<dependency>

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

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

</dependency>

**14. Circuit Breaker: Hystrix Dashboard**

One of the main benefits of Hystrix is the set of metrics it gathers about each HystrixCommand. The Hystrix Dashboard displays the health of each circuit breaker in an efficient manner.

**Figure 14.1. Hystrix Dashboard**



**15. Hystrix Timeouts And Ribbon Clients**

When using Hystrix commands that wrap Ribbon clients you want to make sure your Hystrix timeout is configured to be longer than the configured Ribbon timeout, including any potential retries that might be made. For example, if your Ribbon connection timeout is one second and the Ribbon client might retry the request three times, than your Hystrix timeout should be slightly more than three seconds.

**15.1 How to Include the Hystrix Dashboard**

To include the Hystrix Dashboard in your project, use the starter with a group ID of org.springframework.cloud and an artifact ID of spring-cloud-starter-netflix-hystrix-dashboard. See the [Spring Cloud Project page](https://projects.spring.io/spring-cloud/) for details on setting up your build system with the current Spring Cloud Release Train.

To run the Hystrix Dashboard, annotate your Spring Boot main class with @EnableHystrixDashboard. Then visit /hystrix and point the dashboard to an individual instance’s /hystrix.stream endpoint in a Hystrix client application.

|  |
| --- |
| [Note] |
| When connecting to a /hystrix.stream endpoint that uses HTTPS, the certificate used by the server must be trusted by the JVM. If the certificate is not trusted, you must import the certificate into the JVM in order for the Hystrix Dashboard to make a successful connection to the stream endpoint. |

**15.2 Turbine**

Looking at an individual instance’s Hystrix data is not very useful in terms of the overall health of the system. [Turbine](https://github.com/Netflix/Turbine) is an application that aggregates all of the relevant /hystrix.stream endpoints into a combined /turbine.stream for use in the Hystrix Dashboard. Individual instances are located through Eureka. Running Turbine requires annotating your main class with the @EnableTurbine annotation (for example, by using spring-cloud-starter-netflix-turbine to set up the classpath). All of the documented configuration properties from [the Turbine 1 wiki](https://github.com/Netflix/Turbine/wiki/Configuration-(1.x)) apply. The only difference is that the turbine.instanceUrlSuffix does not need the port prepended, as this is handled automatically unless turbine.instanceInsertPort=false.

|  |
| --- |
| [Note] |
| By default, Turbine looks for the /hystrix.stream endpoint on a registered instance by looking up its hostName and port entries in Eureka and then appending /hystrix.stream to it. If the instance’s metadata contains management.port, it is used instead of the port value for the /hystrix.stream endpoint. By default, the metadata entry called management.port is equal to the management.port configuration property. It can be overridden though with following configuration: |

eureka:

instance:

metadata-map:

management.port: ${management.port:8081}

The turbine.appConfig configuration key is a list of Eureka serviceIds that turbine uses to lookup instances. The turbine stream is then used in the Hystrix dashboard with a URL similar to the following:

<http://my.turbine.server:8080/turbine.stream?cluster=CLUSTERNAME>

The cluster parameter can be omitted if the name is default. The cluster parameter must match an entry in turbine.aggregator.clusterConfig. Values returned from Eureka are upper-case. Consequently, the following example works if there is an application called customers registered with Eureka:

turbine:

aggregator:

clusterConfig: CUSTOMERS

appConfig: customers

If you need to customize which cluster names should be used by Turbine (because you do not want to store cluster names in turbine.aggregator.clusterConfigconfiguration), provide a bean of type TurbineClustersProvider.

The clusterName can be customized by a SPEL expression in turbine.clusterNameExpression with root as an instance of InstanceInfo. The default value is appName, which means that the Eureka serviceId becomes the cluster key (that is, the InstanceInfo for customers has an appName of CUSTOMERS). A different example is turbine.clusterNameExpression=aSGName, which gets the cluster name from the AWS ASG name. The following listing shows another example:

turbine:

aggregator:

clusterConfig: SYSTEM,USER

appConfig: customers,stores,ui,admin

clusterNameExpression: metadata['cluster']

In the preceding example, the cluster name from four services is pulled from their metadata map and is expected to have values that include SYSTEM and USER.

To use the “default” cluster for all apps, you need a string literal expression (with single quotes and escaped with double quotes if it is in YAML as well):

turbine:

appConfig: customers,stores

clusterNameExpression: "'default'"

Spring Cloud provides a spring-cloud-starter-netflix-turbine that has all the dependencies you need to get a Turbine server running. To ad Turnbine, create a Spring Boot application and annotate it with @EnableTurbine.

|  |
| --- |
| [Note] |
| By default, Spring Cloud lets Turbine use the host and port to allow multiple processes per host, per cluster. If you want the native Netflix behavior built into Turbine to *not* allow multiple processes per host, per cluster (the key to the instance ID is the hostname), set turbine.combineHostPort=false. |

**15.2.1 Clusters Endpoint**

In some situations it might be useful for other applications to know what custers have been configured in Turbine. To support this you can use the /clusters endpoint which will return a JSON array of all the configured clusters.

**GET /clusters.**

**[**

**{**

"name": "RACES"**,**

"link": "http://localhost:8383/turbine.stream?cluster=RACES"

**},**

**{**

"name": "WEB"**,**

"link": "http://localhost:8383/turbine.stream?cluster=WEB"

**}**

**]**

This endpoint can be disabled by setting turbine.endpoints.clusters.enabled to false.

**15.3 Turbine Stream**

In some environments (such as in a PaaS setting), the classic Turbine model of pulling metrics from all the distributed Hystrix commands does not work. In that case, you might want to have your Hystrix commands push metrics to Turbine. Spring Cloud enables that with messaging. To do so on the client, add a dependency to spring-cloud-netflix-hystrix-stream and the spring-cloud-starter-stream-\* of your choice. See the [Spring Cloud Stream documentation](https://docs.spring.io/spring-cloud-stream/docs/current/reference/htmlsingle/) for details on the brokers and how to configure the client credentials. It should work out of the box for a local broker.

On the server side, create a Spring Boot application and annotate it with @EnableTurbineStream. The Turbine Stream server requires the use of Spring Webflux, therefore spring-boot-starter-webflux needs to be included in your project. By default spring-boot-starter-webflux is included when adding spring-cloud-starter-netflix-turbine-stream to your application.

You can then point the Hystrix Dashboard to the Turbine Stream Server instead of individual Hystrix streams. If Turbine Stream is running on port 8989 on myhost, then put [http://myhost:8989](http://myhost:8989/) in the stream input field in the Hystrix Dashboard. Circuits are prefixed by their respective serviceId, followed by a dot (.), and then the circuit name.

Spring Cloud provides a spring-cloud-starter-netflix-turbine-stream that has all the dependencies you need to get a Turbine Stream server running. You can then add the Stream binder of your choice — such as spring-cloud-starter-stream-rabbit.

Turbine Stream server also supports the cluster parameter. Unlike Turbine server, Turbine Stream uses eureka serviceIds as cluster names and these are not configurable.

If Turbine Stream server is running on port 8989 on my.turbine.server and you have two eureka serviceIds customers and products in your environment, the following URLs will be available on your Turbine Stream server. default and empty cluster name will provide all metrics that Turbine Stream server receives.

http://my.turbine.sever:8989/turbine.stream?cluster=customers

http://my.turbine.sever:8989/turbine.stream?cluster=products

http://my.turbine.sever:8989/turbine.stream?cluster=default

http://my.turbine.sever:8989/turbine.stream

So, you can use eureka serviceIds as cluster names for your Turbine dashboard (or any compatible dashboard). You don’t need to configure any properties like turbine.appConfig, turbine.clusterNameExpression and turbine.aggregator.clusterConfig for your Turbine Stream server.

|  |
| --- |
| [Note] |
| Turbine Stream server gathers all metrics from the configured input channel with Spring Cloud Stream. It means that it doesn’t gather Hystrix metrics actively from each instance. It just can provide metrics that were already gathered into the input channel by each instance. |

**16. Client Side Load Balancer: Ribbon**

Ribbon is a client-side load balancer that gives you a lot of control over the behavior of HTTP and TCP clients. Feign already uses Ribbon, so, if you use @FeignClient, this section also applies.

A central concept in Ribbon is that of the named client. Each load balancer is part of an ensemble of components that work together to contact a remote server on demand, and the ensemble has a name that you give it as an application developer (for example, by using the @FeignClient annotation). On demand, Spring Cloud creates a new ensemble as an ApplicationContext for each named client by using RibbonClientConfiguration. This contains (amongst other things) an ILoadBalancer, a RestClient, and a ServerListFilter.

**16.1 How to Include Ribbon**

To include Ribbon in your project, use the starter with a group ID of org.springframework.cloud and an artifact ID of spring-cloud-starter-netflix-ribbon. See the [Spring Cloud Project page](https://projects.spring.io/spring-cloud/) for details on setting up your build system with the current Spring Cloud Release Train.

**16.2 Customizing the Ribbon Client**

You can configure some bits of a Ribbon client by using external properties in <client>.ribbon.\*, which is similar to using the Netflix APIs natively, except that you can use Spring Boot configuration files. The native options can be inspected as static fields in [CommonClientConfigKey](https://github.com/Netflix/ribbon/blob/master/ribbon-core/src/main/java/com/netflix/client/config/CommonClientConfigKey.java) (part of ribbon-core).

Spring Cloud also lets you take full control of the client by declaring additional configuration (on top of the RibbonClientConfiguration) using @RibbonClient, as shown in the following example:

*@Configuration*

*@RibbonClient(name = "custom", configuration = CustomConfiguration.class)*

**public** **class** TestConfiguration {

}

In this case, the client is composed from the components already in RibbonClientConfiguration, together with any in CustomConfiguration (where the latter generally overrides the former).

|  |
| --- |
| [Warning] |
| The CustomConfiguration clas must be a @Configuration class, but take care that it is not in a @ComponentScan for the main application context. Otherwise, it is shared by all the @RibbonClients. If you use @ComponentScan (or @SpringBootApplication), you need to take steps to avoid it being included (for instance, you can put it in a separate, non-overlapping package or specify the packages to scan explicitly in the @ComponentScan). |

The following table shows the beans that Spring Cloud Netflix provides by default for Ribbon:

| **Bean Type** | **Bean Name** | **Class Name** |
| --- | --- | --- |
| IClientConfig | ribbonClientConfig | DefaultClientConfigImpl |
| IRule | ribbonRule | ZoneAvoidanceRule |
| IPing | ribbonPing | DummyPing |
| ServerList<Server> | ribbonServerList | ConfigurationBasedServerList |
| ServerListFilter<Server> | ribbonServerListFilter | ZonePreferenceServerListFilter |
| ILoadBalancer | ribbonLoadBalancer | ZoneAwareLoadBalancer |
| ServerListUpdater | ribbonServerListUpdater | PollingServerListUpdater |

Creating a bean of one of those type and placing it in a @RibbonClient configuration (such as FooConfiguration above) lets you override each one of the beans described, as shown in the following example:

*@Configuration*

**protected** **static** **class** FooConfiguration {

*@Bean*

**public** ZonePreferenceServerListFilter serverListFilter() {

ZonePreferenceServerListFilter filter = **new** ZonePreferenceServerListFilter();

filter.setZone("myTestZone");

**return** filter;

}

*@Bean*

**public** IPing ribbonPing() {

**return** **new** PingUrl();

}

}

The include statement in the preceding example replaces NoOpPing with PingUrl and provides a custom serverListFilter.

**16.3 Customizing the Default for All Ribbon Clients**

A default configuration can be provided for all Ribbon Clients by using the @RibbonClients annotation and registering a default configuration, as shown in the following example:

*@RibbonClients(defaultConfiguration = DefaultRibbonConfig.class)*

**public** **class** RibbonClientDefaultConfigurationTestsConfig {

**public** **static** **class** BazServiceList **extends** ConfigurationBasedServerList {

**public** BazServiceList(IClientConfig config) {

**super**.initWithNiwsConfig(config);

}

}

}

*@Configuration*

**class** DefaultRibbonConfig {

*@Bean*

**public** IRule ribbonRule() {

**return** **new** BestAvailableRule();

}

*@Bean*

**public** IPing ribbonPing() {

**return** **new** PingUrl();

}

*@Bean*

**public** ServerList<Server> ribbonServerList(IClientConfig config) {

**return** **new** RibbonClientDefaultConfigurationTestsConfig.BazServiceList(config);

}

*@Bean*

**public** ServerListSubsetFilter serverListFilter() {

ServerListSubsetFilter filter = **new** ServerListSubsetFilter();

**return** filter;

}

}

**16.4 Customizing the Ribbon Client by Setting Properties**

Starting with version 1.2.0, Spring Cloud Netflix now supports customizing Ribbon clients by setting properties to be compatible with the [Ribbon documentation](https://github.com/Netflix/ribbon/wiki/Working-with-load-balancers#components-of-load-balancer).

This lets you change behavior at start up time in different environments.

The following list shows the supported properties>:

* <clientName>.ribbon.NFLoadBalancerClassName: Should implement ILoadBalancer
* <clientName>.ribbon.NFLoadBalancerRuleClassName: Should implement IRule
* <clientName>.ribbon.NFLoadBalancerPingClassName: Should implement IPing
* <clientName>.ribbon.NIWSServerListClassName: Should implement ServerList
* <clientName>.ribbon.NIWSServerListFilterClassName: Should implement ServerListFilter

|  |
| --- |
| [Note] |
| Classes defined in these properties have precedence over beans defined by using @RibbonClient(configuration=MyRibbonConfig.class) and the defaults provided by Spring Cloud Netflix. |

To set the IRule for a service name called users, you could set the following properties:

**application.yml.**

users:

ribbon:

NIWSServerListClassName: com.netflix.loadbalancer.ConfigurationBasedServerList

NFLoadBalancerRuleClassName: com.netflix.loadbalancer.WeightedResponseTimeRule

See the [Ribbon documentation](https://github.com/Netflix/ribbon/wiki/Working-with-load-balancers) for implementations provided by Ribbon.

**16.5 Using Ribbon with Eureka**

When Eureka is used in conjunction with Ribbon (that is, both are on the classpath), the ribbonServerList is overridden with an extension of DiscoveryEnabledNIWSServerList, which populates the list of servers from Eureka. It also replaces the IPing interface with NIWSDiscoveryPing, which delegates to Eureka to determine if a server is up. The ServerList that is installed by default is a DomainExtractingServerList. Its purpose is to make metadata available to the load balancer without using AWS AMI metadata (which is what Netflix relies on). By default, the server list is constructed with “zone” information, as provided in the instance metadata (so, on the remote clients, set eureka.instance.metadataMap.zone). If that is missing and if the approximateZoneFromHostname flag is set, it can use the domain name from the server hostname as a proxy for the zone. Once the zone information is available, it can be used in a ServerListFilter. By default, it is used to locate a server in the same zone as the client, because the default is a ZonePreferenceServerListFilter. By default, the zone of the client is determined in the same way as the remote instances (that is, through eureka.instance.metadataMap.zone).

|  |
| --- |
| [Note] |
| The orthodox “archaius” way to set the client zone is through a configuration property called "@zone". If it is available, Spring Cloud uses that in preference to all other settings (note that the key must be quoted in YAML configuration). |
| [Note] |
| If there is no other source of zone data, then a guess is made, based on the client configuration (as opposed to the instance configuration). We take eureka.client.availabilityZones, which is a map from region name to a list of zones, and pull out the first zone for the instance’s own region (that is, the eureka.client.region, which defaults to "us-east-1", for compatibility with native Netflix). | |

**16.6 Example: How to Use Ribbon Without Eureka**

Eureka is a convenient way to abstract the discovery of remote servers so that you do not have to hard code their URLs in clients. However, if you prefer not to use Eureka, Ribbon and Feign also work. Suppose you have declared a @RibbonClient for "stores", and Eureka is not in use (and not even on the classpath). The Ribbon client defaults to a configured server list. You can supply the configuration as follows:

**application.yml.**

stores:

ribbon:

listOfServers: example.com,google.com

**16.7 Example: Disable Eureka Use in Ribbon**

Setting the ribbon.eureka.enabled property to false explicitly disables the use of Eureka in Ribbon, as shown in the following example:

**application.yml.**

ribbon:

eureka:

enabled: false

**16.8 Using the Ribbon API Directly**

You can also use the LoadBalancerClient directly, as shown in the following example:

**public** **class** MyClass {

*@Autowired*

**private** LoadBalancerClient loadBalancer;

**public** **void** doStuff() {

ServiceInstance instance = loadBalancer.choose("stores");

URI storesUri = URI.create(String.format("http://%s:%s", instance.getHost(), instance.getPort()));

*// ... do something with the URI*

}

}

**16.9 Caching of Ribbon Configuration**

Each Ribbon named client has a corresponding child application Context that Spring Cloud maintains. This application context is lazily loaded on the first request to the named client. This lazy loading behavior can be changed to instead eagerly load these child application contexts at startup, by specifying the names of the Ribbon clients, as shown in the following example:

**application.yml.**

ribbon:

eager-load:

enabled: true

clients: client1, client2, client3

**16.10 How to Configure Hystrix Thread Pools**

If you change zuul.ribbonIsolationStrategy to THREAD, the thread isolation strategy for Hystrix is used for all routes. In that case, the HystrixThreadPoolKey is set to RibbonCommand as the default. It means that HystrixCommands for all routes are executed in the same Hystrix thread pool. This behavior can be changed with the following configuration:

**application.yml.**

zuul:

threadPool:

useSeparateThreadPools: true

The preceding example results in HystrixCommands being executed in the Hystrix thread pool for each route.

In this case, the default HystrixThreadPoolKey is the same as the service ID for each route. To add a prefix to HystrixThreadPoolKey, set zuul.threadPool.threadPoolKeyPrefix to the value that you want to add, as shown in the following example:

**application.yml.**

zuul:

threadPool:

useSeparateThreadPools: true

threadPoolKeyPrefix: zuulgw

**16.11 How to Provide a Key to Ribbon’s IRule**

If you need to provide your own IRule implementation to handle a special routing requirement like a “canary” test, pass some information to the choose method of IRule.

**com.netflix.loadbalancer.IRule.java.**

public interface IRule{

public Server choose(Object key);

:

You can provide some information that is used by your IRule implementation to choose a target server, as shown in the following example:

RequestContext.getCurrentContext()

.set(FilterConstants.LOAD\_BALANCER\_KEY, "canary-test");

If you put any object into the RequestContext with a key of FilterConstants.LOAD\_BALANCER\_KEY, it is passed to the choose method of the IRuleimplementation. The code shown in the preceding example must be executed before RibbonRoutingFilter is executed. Zuul’s pre filter is the best place to do that. You can access HTTP headers and query parameters through the RequestContext in pre filter, so it can be used to determine the LOAD\_BALANCER\_KEY that is passed to Ribbon. If you do not put any value with LOAD\_BALANCER\_KEY in RequestContext, null is passed as a parameter of the choose method.

**17. External Configuration: Archaius**

[Archaius](https://github.com/Netflix/archaius) is the Netflix client-side configuration library. It is the library used by all of the Netflix OSS components for configuration. Archaius is an extension of the [Apache Commons Configuration](https://commons.apache.org/proper/commons-configuration) project. It allows updates to configuration by either polling a source for changes or by letting a source push changes to the client. Archaius uses Dynamic<Type>Property classes as handles to properties, as shown in the following example:

**Archaius Example.**

**class** ArchaiusTest {

DynamicStringProperty myprop = DynamicPropertyFactory

.getInstance()

.getStringProperty("my.prop");

**void** doSomething() {

OtherClass.someMethod(myprop.get());

}

}

Archaius has its own set of configuration files and loading priorities. Spring applications should generally not use Archaius directly, but the need to configure the Netflix tools natively remains. Spring Cloud has a Spring Environment Bridge so that Archaius can read properties from the Spring Environment. This bridge allows Spring Boot projects to use the normal configuration toolchain while letting them configure the Netflix tools as documented (for the most part).

**18. Router and Filter: Zuul**

Routing is an integral part of a microservice architecture. For example, / may be mapped to your web application, /api/users is mapped to the user service and /api/shop is mapped to the shop service. [Zuul](https://github.com/Netflix/zuul) is a JVM-based router and server-side load balancer from Netflix.

[Netflix uses Zuul](http://www.slideshare.net/MikeyCohen1/edge-architecture-ieee-international-conference-on-cloud-engineering-32240146/27) for the following:

* Authentication
* Insights
* Stress Testing
* Canary Testing
* Dynamic Routing
* Service Migration
* Load Shedding
* Security
* Static Response handling
* Active/Active traffic management

Zuul’s rule engine lets rules and filters be written in essentially any JVM language, with built-in support for Java and Groovy.

|  |
| --- |
| [Note] |
| The configuration property zuul.max.host.connections has been replaced by two new properties, zuul.host.maxTotalConnections and zuul.host.maxPerRouteConnections, which default to 200 and 20 respectively. | |
| [Note] | |
| The default Hystrix isolation pattern (ExecutionIsolationStrategy) for all routes is SEMAPHORE. zuul.ribbonIsolationStrategy can be changed to THREAD if that isolation pattern is preferred. | |

**18.1 How to Include Zuul**

To include Zuul in your project, use the starter with a group ID of org.springframework.cloud and a artifact ID of spring-cloud-starter-netflix-zuul. See the [Spring Cloud Project page](https://projects.spring.io/spring-cloud/) for details on setting up your build system with the current Spring Cloud Release Train.

**18.2 Embedded Zuul Reverse Proxy**

Spring Cloud has created an embedded Zuul proxy to ease the development of a common use case where a UI application wants to make proxy calls to one or more back end services. This feature is useful for a user interface to proxy to the back end services it requires, avoiding the need to manage CORS and authentication concerns independently for all the back ends.

To enable it, annotate a Spring Boot main class with @EnableZuulProxy. Doing so causes local calls to be forwarded to the appropriate service. By convention, a service with an ID of users receives requests from the proxy located at /users (with the prefix stripped). The proxy uses Ribbon to locate an instance to which to forward through discovery. All requests are executed in a [hystrix command](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#hystrix-fallbacks-for-routes), so failures appear in Hystrix metrics. Once the circuit is open, the proxy does not try to contact the service.

|  |
| --- |
| [Note] |
| the Zuul starter does not include a discovery client, so, for routes based on service IDs, you need to provide one of those on the classpath as well (Eureka is one choice). |

To skip having a service automatically added, set zuul.ignored-services to a list of service ID patterns. If a service matches a pattern that is ignored but is also included in the explicitly configured routes map, it is unignored, as shown in the following example:

**application.yml.**

zuul:

ignoredServices: '\*'

routes:

users: /myusers/\*\*

In the preceding example, all services are ignored, **except** for users.

To augment or change the proxy routes, you can add external configuration, as follows:

**application.yml.**

zuul:

routes:

users: /myusers/\*\*

The preceding example means that HTTP calls to /myusers get forwarded to the users service (for example /myusers/101 is forwarded to /101).

To get more fine-grained control over a route, you can specify the path and the serviceId independently, as follows:

**application.yml.**

zuul:

routes:

users:

path: /myusers/\*\*

serviceId: users\_service

The preceding example means that HTTP calls to /myusers get forwarded to the users\_service service. The route must have a path that can be specified as an ant-style pattern, so /myusers/\* only matches one level, but /myusers/\*\* matches hierarchically.

The location of the back end can be specified as either a serviceId (for a service from discovery) or a url (for a physical location), as shown in the following example:

**application.yml.**

zuul:

routes:

users:

path: /myusers/\*\*

url: http://example.com/users\_service

These simple url-routes do not get executed as a HystrixCommand, nor do they load-balance multiple URLs with Ribbon. To achieve those goals, you can specify a serviceId with a static list of servers, as follows:

**application.yml.**

zuul:

routes:

echo:

path: /myusers/\*\*

serviceId: myusers-service

stripPrefix: **true**

hystrix:

command:

myusers-service:

execution:

isolation:

thread:

timeoutInMilliseconds: ...

myusers-service:

ribbon:

NIWSServerListClassName: com.netflix.loadbalancer.ConfigurationBasedServerList

listOfServers: http://example1.com,http://example2.com

ConnectTimeout: 1000

ReadTimeout: 3000

MaxTotalHttpConnections: 500

MaxConnectionsPerHost: 100

Another method is specifiying a service-route and configuring a Ribbon client for the serviceId (doing so requires disabling Eureka support in Ribbon — see [above for more information](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-ribbon-without-eureka)), as shown in the following example:

**application.yml.**

zuul:

routes:

users:

path: /myusers/\*\*

serviceId: users

ribbon:

eureka:

enabled: **false**

users:

ribbon:

listOfServers: example.com,google.com

You can provide a convention between serviceId and routes by using regexmapper. It uses regular-expression named groups to extract variables from serviceIdand inject them into a route pattern, as shown in the following example:

**ApplicationConfiguration.java.**

*@Bean*

**public** PatternServiceRouteMapper serviceRouteMapper() {

**return** **new** PatternServiceRouteMapper(

"(?<name>^.+)-(?<version>v.+$)",

"${version}/${name}");

}

The preceding example means that a serviceId of myusers-v1 is mapped to route /v1/myusers/\*\*. Any regular expression is accepted, but all named groups must be present in both servicePattern and routePattern. If servicePattern does not match a serviceId, the default behavior is used. In the preceding example, a serviceId of myusers is mapped to the "/myusers/\*\*" route (with no version detected). This feature is disabled by default and only applies to discovered services.

To add a prefix to all mappings, set zuul.prefix to a value, such as /api. By default, the proxy prefix is stripped from the request before the request is forwarded by (you can switch this behavior off with zuul.stripPrefix=false). You can also switch off the stripping of the service-specific prefix from individual routes, as shown in the following example:

**application.yml.**

zuul:

routes:

users:

path: /myusers/\*\*

stripPrefix: **false**

|  |
| --- |
| [Note] |
| zuul.stripPrefix only applies to the prefix set in zuul.prefix. It does not have any effect on prefixes defined within a given route’s path. |

In the preceding example, requests to /myusers/101 are forwarded to /myusers/101 on the users service.

The zuul.routes entries actually bind to an object of type ZuulProperties. If you look at the properties of that object, you can see that it also has a retryable flag. Set that flag to true to have the Ribbon client automatically retry failed requests. You can also set that flag to true when you need to modify the parameters of the retry operations that use the Ribbon client configuration.

By default, the X-Forwarded-Host header is added to the forwarded requests. To turn it off, set zuul.addProxyHeaders = false. By default, the prefix path is stripped, and the request to the back end picks up a X-Forwarded-Prefix header (/myusers in the examples shown earlier).

If you set a default route (/), an application with @EnableZuulProxy could act as a standalone server. For example, zuul.route.home: / would route all traffic ("/\*\*") to the "home" service.

If more fine-grained ignoring is needed, you can specify specific patterns to ignore. These patterns are evaluated at the start of the route location process, which means prefixes should be included in the pattern to warrant a match. Ignored patterns span all services and supersede any other route specification. The following example shows how to create ignored patterns:

**application.yml.**

zuul:

ignoredPatterns: /\*\*/admin/\*\*

routes:

users: /myusers/\*\*

The preceding example means that all calls (such as /myusers/101) are forwarded to /101 on the users service. However, calls including /admin/ do not resolve.

|  |
| --- |
| [Warning] |
| If you need your routes to have their order preserved, you need to use a YAML file, as the ordering is lost when using a properties file. The following example shows such a YAML file: |

**application.yml.**

zuul:

routes:

users:

path: /myusers/\*\*

legacy:

path: /\*\*

If you were to use a properties file, the legacy path might end up in front of the users path, rendering the users path unreachable.

**18.3 Zuul Http Client**

The default HTTP client used by Zuul is now backed by the Apache HTTP Client instead of the deprecated Ribbon RestClient. To use RestClient or okhttp3.OkHttpClient, set ribbon.restclient.enabled=true or ribbon.okhttp.enabled=true, respectively. If you would like to customize the Apache HTTP client or the OK HTTP client, provide a bean of type ClosableHttpClient or OkHttpClient.

**18.4 Cookies and Sensitive Headers**

You can share headers between services in the same system, but you probably do not want sensitive headers leaking downstream into external servers. You can specify a list of ignored headers as part of the route configuration. Cookies play a special role, because they have well defined semantics in browsers, and they are always to be treated as sensitive. If the consumer of your proxy is a browser, then cookies for downstream services also cause problems for the user, because they all get jumbled up together (all downstream services look like they come from the same place).

If you are careful with the design of your services, (for example, if only one of the downstream services sets cookies), you might be able to let them flow from the back end all the way up to the caller. Also, if your proxy sets cookies and all your back-end services are part of the same system, it can be natural to simply share them (and, for instance, use Spring Session to link them up to some shared state). Other than that, any cookies that get set by downstream services are likely to be not useful to the caller, so it is recommended that you make (at least) Set-Cookie and Cookie into sensitive headers for routes that are not part of your domain. Even for routes that are part of your domain, try to think carefully about what it means before letting cookies flow between them and the proxy.

The sensitive headers can be configured as a comma-separated list per route, as shown in the following example:

**application.yml.**

zuul:

routes:

users:

path: /myusers/\*\*

sensitiveHeaders: Cookie,Set-Cookie,Authorization

url: https://downstream

|  |
| --- |
| [Note] |
| This is the default value for sensitiveHeaders, so you need not set it unless you want it to be different. This is new in Spring Cloud Netflix 1.1 (in 1.0, the user had no control over headers, and all cookies flowed in both directions). |

The sensitiveHeaders are a blacklist, and the default is not empty. Consequently, to make Zuul send all headers (except the ignored ones), you must explicitly set it to the empty list. Doing so is necessary if you want to pass cookie or authorization headers to your back end. The following example shows how to use sensitiveHeaders:

**application.yml.**

zuul:

routes:

users:

path: /myusers/\*\*

sensitiveHeaders:

url: https://downstream

You can also set sensitive headers, by setting zuul.sensitiveHeaders. If sensitiveHeaders is set on a route, it overrides the global sensitiveHeaders setting.

**18.5 Ignored Headers**

In addition to the route-sensitive headers, you can set a global value called zuul.ignoredHeaders for values (both request and response) that should be discarded during interactions with downstream services. By default, if Spring Security is not on the classpath, these are empty. Otherwise, they are initialized to a set of well known “security” headers (for example, involving caching) as specified by Spring Security. The assumption in this case is that the downstream services might add these headers, too, but we want the values from the proxy. To not discard these well known security headers when Spring Security is on the classpath, you can set zuul.ignoreSecurityHeaders to false. Doing so can be useful if you disabled the HTTP Security response headers in Spring Security and want the values provided by downstream services.

**18.6 Management Endpoints**

By default, if you use @EnableZuulProxy with the Spring Boot Actuator, you enable two additional endpoints:

* Routes
* Filters

**18.6.1 Routes Endpoint**

A GET to the routes endpoint at /routes returns a list of the mapped routes:

**GET /routes.**

**{**

/stores/\*\*: "http://localhost:8081"

**}**

Additional route details can be requested by adding the ?format=details query string to /routes. Doing so produces the following output:

**GET /routes/details.**

**{**

"/stores/\*\*": **{**

"id": "stores"**,**

"fullPath": "/stores/\*\*"**,**

"location": "http://localhost:8081"**,**

"path": "/\*\*"**,**

"prefix": "/stores"**,**

"retryable": **false,**

"customSensitiveHeaders": **false,**

"prefixStripped": **true**

**}**

**}**

A POST to /routes forces a refresh of the existing routes (for example, when there have been changes in the service catalog). You can disable this endpoint by setting endpoints.routes.enabled to false.

|  |
| --- |
| [Note] |
| the routes should respond automatically to changes in the service catalog, but the POST to /routes is a way to force the change to happen immediately. |

**18.6.2 Filters Endpoint**

A GET to the filters endpoint at /filters returns a map of Zuul filters by type. For each filter type in the map, you get a list of all the filters of that type, along with their details.

**18.7 Strangulation Patterns and Local Forwards**

A common pattern when migrating an existing application or API is to “strangle” old endpoints, slowly replacing them with different implementations. The Zuul proxy is a useful tool for this because you can use it to handle all traffic from the clients of the old endpoints but redirect some of the requests to new ones.

The following example shows the configuration details for a “strangle” scenario:

**application.yml.**

zuul:

routes:

first:

path: /first/\*\*

url: http://first.example.com

second:

path: /second/\*\*

url: forward:/second

third:

path: /third/\*\*

url: forward:/3rd

legacy:

path: /\*\*

url: http://legacy.example.com

In the preceding example, we are strangle the “legacy” application, which is mapped to all requests that do not match one of the other patterns. Paths in /first/\*\*have been extracted into a new service with an external URL. Paths in /second/\*\* are forwarded so that they can be handled locally (for example, with a normal Spring @RequestMapping). Paths in /third/\*\* are also forwarded but with a different prefix (/third/foo is forwarded to /3rd/foo).

|  |
| --- |
| [Note] |
| The ignored patterns aren’t completely ignored, they just are not handled by the proxy (so they are also effectively forwarded locally). |

**18.8 Uploading Files through Zuul**

If you use @EnableZuulProxy, you can use the proxy paths to upload files and it should work, so long as the files are small. For large files there is an alternative path that bypasses the Spring DispatcherServlet (to avoid multipart processing) in "/zuul/\*". In other words, if you have zuul.routes.customers=/customers/\*\*, then you can POST large files to /zuul/customers/\*. The servlet path is externalized via zuul.servletPath. If the proxy route takes you through a Ribbon load balancer, extremely large files also require elevated timeout settings, as shown in the following example:

**application.yml.**

hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds: 60000

ribbon:

ConnectTimeout: 3000

ReadTimeout: 60000

Note that, for streaming to work with large files, you need to use chunked encoding in the request (which some browsers do not do by default), as shown in the following example:

$ curl -v -H "Transfer-Encoding: chunked" \

-F "file=@mylarge.iso" localhost:9999/zuul/simple/file

**18.9 Query String Encoding**

When processing the incoming request, query params are decoded so that they can be available for possible modifications in Zuul filters. They are then re-encoded the back end request is rebuilt in the route filters. The result can be different than the original input if (for example) it was encoded with Javascript’s encodeURIComponent()method. While this causes no issues in most cases, some web servers can be picky with the encoding of complex query string.

To force the original encoding of the query string, it is possible to pass a special flag to ZuulProperties so that the query string is taken as is with the HttpServletRequest::getQueryString method, as shown in the following example:

**application.yml.**

zuul:

forceOriginalQueryStringEncoding: **true**

|  |
| --- |
| [Note] |
| This special flag works only with SimpleHostRoutingFilter. Also, you loose the ability to easily override query parameters with RequestContext.getCurrentContext().setRequestQueryParams(someOverriddenParameters), because the query string is now fetched directly on the original HttpServletRequest. |

**18.10 Plain Embedded Zuul**

If you use @EnableZuulServer (instead of @EnableZuulProxy), you can also run a Zuul server without proxying or selectively switch on parts of the proxying platform. Any beans that you add to the application of type ZuulFilter are installed automatically (as they are with @EnableZuulProxy) but without any of the proxy filters being added automatically.

In that case, the routes into the Zuul server are still specified by configuring "zuul.routes.\*", but there is no service discovery and no proxying. Consequently, the "serviceId" and "url" settings are ignored. The following example maps all paths in "/api/\*\*" to the Zuul filter chain:

**application.yml.**

zuul:

routes:

api: /api/\*\*

**18.11 Disable Zuul Filters**

Zuul for Spring Cloud comes with a number of ZuulFilter beans enabled by default in both proxy and server mode. See [the Zuul filters package](https://github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-cloud-netflix-zuul/src/main/java/org/springframework/cloud/netflix/zuul/filters) for the list of filters that you can enable. If you want to disable one, set zuul.<SimpleClassName>.<filterType>.disable=true. By convention, the package after filters is the Zuul filter type. For example to disable org.springframework.cloud.netflix.zuul.filters.post.SendResponseFilter, set zuul.SendResponseFilter.post.disable=true.

**18.12 Providing Hystrix Fallbacks For Routes**

When a circuit for a given route in Zuul is tripped, you can provide a fallback response by creating a bean of type FallbackProvider. Within this bean, you need to specify the route ID the fallback is for and provide a ClientHttpResponse to return as a fallback. The following example shows a relatively simple FallbackProviderimplementation:

**class** MyFallbackProvider **implements** FallbackProvider {

*@Override*

**public** String getRoute() {

**return** "customers";

}

*@Override*

**public** ClientHttpResponse fallbackResponse(String route, **final** Throwable cause) {

**if** (cause **instanceof** HystrixTimeoutException) {

**return** response(HttpStatus.GATEWAY\_TIMEOUT);

} **else** {

**return** response(HttpStatus.INTERNAL\_SERVER\_ERROR);

}

}

**private** ClientHttpResponse response(**final** HttpStatus status) {

**return** **new** ClientHttpResponse() {

*@Override*

**public** HttpStatus getStatusCode() **throws** IOException {

**return** status;

}

*@Override*

**public** **int** getRawStatusCode() **throws** IOException {

**return** status.value();

}

*@Override*

**public** String getStatusText() **throws** IOException {

**return** status.getReasonPhrase();

}

*@Override*

**public** **void** close() {

}

*@Override*

**public** InputStream getBody() **throws** IOException {

**return** **new** ByteArrayInputStream("fallback".getBytes());

}

*@Override*

**public** HttpHeaders getHeaders() {

HttpHeaders headers = **new** HttpHeaders();

headers.setContentType(MediaType.APPLICATION\_JSON);

**return** headers;

}

};

}

}

The following example shows how the route configuration for the previous example might appear:

zuul:

routes:

customers: /customers/\*\*

If you would like to provide a default fallback for all routes, you can create a bean of type FallbackProvider and have the getRoute method return \* or null, as shown in the following example:

**class** MyFallbackProvider **implements** FallbackProvider {

*@Override*

**public** String getRoute() {

**return** "\*";

}

*@Override*

**public** ClientHttpResponse fallbackResponse(String route, Throwable throwable) {

**return** **new** ClientHttpResponse() {

*@Override*

**public** HttpStatus getStatusCode() **throws** IOException {

**return** HttpStatus.OK;

}

*@Override*

**public** **int** getRawStatusCode() **throws** IOException {

**return** 200;

}

*@Override*

**public** String getStatusText() **throws** IOException {

**return** "OK";

}

*@Override*

**public** **void** close() {

}

*@Override*

**public** InputStream getBody() **throws** IOException {

**return** **new** ByteArrayInputStream("fallback".getBytes());

}

*@Override*

**public** HttpHeaders getHeaders() {

HttpHeaders headers = **new** HttpHeaders();

headers.setContentType(MediaType.APPLICATION\_JSON);

**return** headers;

}

};

}

}

**18.13 Zuul Timeouts**

If you want to configure the socket timeouts and read timeouts for requests proxied through Zuul, you have two options, based on your configuration:

* If Zuul uses service discovery, you need to configure these timeouts with the ribbon.ReadTimeout and ribbon.SocketTimeout Ribbon properties.

If you have configured Zuul routes by specifying URLs, you need to use zuul.host.connect-timeout-millis and zuul.host.socket-timeout-millis.

**18.14 Rewriting the Location header**

If Zuul is fronting a web application, you may need to re-write the Location header when the web application redirects through a HTTP status code of 3XX. Otherwise, the browser redirects to the web application’s URL instead of the Zuul URL. You can configure a LocationRewriteFilter Zuul filter to re-write the Location header to the Zuul’s URL. It also adds back the stripped global and route-specific prefixes. The following example adds a filter by using a Spring Configuration file:

**import** org.springframework.cloud.netflix.zuul.filters.post.LocationRewriteFilter;

...

*@Configuration*

*@EnableZuulProxy*

**public** **class** ZuulConfig {

*@Bean*

**public** LocationRewriteFilter locationRewriteFilter() {

**return** **new** LocationRewriteFilter();

}

}

|  |  |
| --- | --- |
| [Caution] | **Caution** |
| Use this filter carefully. The filter acts on the Location header of ALL 3XX response codes, which may not be appropriate in all scenarios, such as when redirecting the user to an external URL. |

**18.15 Metrics**

Zuul will provide metrics under the Actuator metrics endpoint for any failures that might occur when routing requests. These metrics can be viewed by hitting /actuator/metrics. The metrics will have a name that has the format ZUUL::EXCEPTION:errorCause:statusCode.

**18.16 Zuul Developer Guide**

For a general overview of how Zuul works, see [the Zuul Wiki](https://github.com/Netflix/zuul/wiki/How-it-Works).

**18.16.1 The Zuul Servlet**

Zuul is implemented as a Servlet. For the general cases, Zuul is embedded into the Spring Dispatch mechanism. This lets Spring MVC be in control of the routing. In this case, Zuul buffers requests. If there is a need to go through Zuul without buffering requests (for example, for large file uploads), the Servlet is also installed outside of the Spring Dispatcher. By default, the servlet has an address of /zuul. This path can be changed with the zuul.servlet-path property.

**18.16.2 Zuul RequestContext**

To pass information between filters, Zuul uses a [RequestContext](https://github.com/Netflix/zuul/blob/1.x/zuul-core/src/main/java/com/netflix/zuul/context/RequestContext.java). Its data is held in a ThreadLocal specific to each request. Information about where to route requests, errors, and the actual HttpServletRequest and HttpServletResponse are stored there. The RequestContext extends ConcurrentHashMap, so anything can be stored in the context. [FilterConstants](https://github.com/spring-cloud/spring-cloud-netflix/blob/master/spring-cloud-netflix-core/src/main/java/org/springframework/cloud/netflix/zuul/filters/support/FilterConstants.java) contains the keys used by the filters installed by Spring Cloud Netflix (more on these [later](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#zuul-developer-guide-enable-filters)).

**18.16.3 @EnableZuulProxy vs. @EnableZuulServer**

Spring Cloud Netflix installs a number of filters, depending on which annotation was used to enable Zuul. @EnableZuulProxy is a superset of @EnableZuulServer. In other words, @EnableZuulProxy contains all the filters installed by @EnableZuulServer. The additional filters in the “proxy” enable routing functionality. If you want a “blank” Zuul, you should use @EnableZuulServer.

**18.16.4 @EnableZuulServer Filters**

@EnableZuulServer creates a SimpleRouteLocator that loads route definitions from Spring Boot configuration files.

The following filters are installed (as normal Spring Beans):

* Pre filters:
  + ServletDetectionFilter: Detects whether the request is through the Spring Dispatcher. Sets a boolean with a key of FilterConstants.IS\_DISPATCHER\_SERVLET\_REQUEST\_KEY.
  + FormBodyWrapperFilter: Parses form data and re-encodes it for downstream requests.
  + DebugFilter: If the debug request parameter is set, sets RequestContext.setDebugRouting() and RequestContext.setDebugRequest() to true. \*Route filters:
  + SendForwardFilter: Forwards requests by using the Servlet RequestDispatcher. The forwarding location is stored in the RequestContext attribute, FilterConstants.FORWARD\_TO\_KEY. This is useful for forwarding to endpoints in the current application.
* Post filters:
  + SendResponseFilter: Writes responses from proxied requests to the current response.
* Error filters:
  + SendErrorFilter: Forwards to /error (by default) if RequestContext.getThrowable() is not null. You can change the default forwarding path (/error) by setting the error.path property.

**18.16.5 @EnableZuulProxy Filters**

Creates a DiscoveryClientRouteLocator that loads route definitions from a DiscoveryClient (such as Eureka) as well as from properties. A route is created for each serviceId from the DiscoveryClient. As new services are added, the routes are refreshed.

In addition to the filters described earlier, the following filters are installed (as normal Spring Beans):

* Pre filters:
  + PreDecorationFilter: Determines where and how to route, depending on the supplied RouteLocator. It also sets various proxy-related headers for downstream requests.
* Route filters:
  + RibbonRoutingFilter: Uses Ribbon, Hystrix, and pluggable HTTP clients to send requests. Service IDs are found in the RequestContext attribute, FilterConstants.SERVICE\_ID\_KEY. This filter can use different HTTP clients:
    - Apache HttpClient: The default client.
    - Squareup OkHttpClient v3: Enabled by having the com.squareup.okhttp3:okhttp library on the classpath and setting ribbon.okhttp.enabled=true.
    - Netflix Ribbon HTTP client: Enabled by setting ribbon.restclient.enabled=true. This client has limitations, including that it does not support the PATCH method, but it also has built-in retry.
  + SimpleHostRoutingFilter: Sends requests to predetermined URLs through an Apache HttpClient. URLs are found in RequestContext.getRouteHost().

**18.16.6 Custom Zuul Filter Examples**

Most of the following "How to Write" examples below are included [Sample Zuul Filters](https://github.com/spring-cloud-samples/sample-zuul-filters) project. There are also examples of manipulating the request or response body in that repository.

This section includes the following examples:

* [the section called “How to Write a Pre Filter”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#zuul-developer-guide-sample-pre-filter)
* [the section called “How to Write a Route Filter”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#zuul-developer-guide-sample-route-filter)
* [the section called “How to Write a Post Filter”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#zuul-developer-guide-sample-post-filter)

**How to Write a Pre Filter**

Pre filters set up data in the RequestContext for use in filters downstream. The main use case is to set information required for route filters. The following example shows a Zuul pre filter:

**public** **class** QueryParamPreFilter **extends** ZuulFilter {

*@Override*

**public** **int** filterOrder() {

**return** PRE\_DECORATION\_FILTER\_ORDER - 1; *// run before PreDecoration*

}

*@Override*

**public** String filterType() {

**return** PRE\_TYPE;

}

*@Override*

**public** **boolean** shouldFilter() {

RequestContext ctx = RequestContext.getCurrentContext();

**return** !ctx.containsKey(FORWARD\_TO\_KEY) *// a filter has already forwarded*

&& !ctx.containsKey(SERVICE\_ID\_KEY); *// a filter has already determined serviceId*

}

*@Override*

**public** Object run() {

RequestContext ctx = RequestContext.getCurrentContext();

HttpServletRequest request = ctx.getRequest();

**if** (request.getParameter("sample") != null) {

*// put the serviceId in `RequestContext`*

ctx.put(SERVICE\_ID\_KEY, request.getParameter("foo"));

}

**return** null;

}

}

The preceding filter populates SERVICE\_ID\_KEY from the sample request parameter. In practice, you should not do that kind of direct mapping. Instead, the service ID should be looked up from the value of sample instead.

Now that SERVICE\_ID\_KEY is populated, PreDecorationFilter does not run and RibbonRoutingFilter runs.

|  |
| --- |
| [Tip] |
| If you want to route to a full URL, call ctx.setRouteHost(url) instead. |

To modify the path to which routing filters forward, set the REQUEST\_URI\_KEY.

**How to Write a Route Filter**

Route filters run after pre filters and make requests to other services. Much of the work here is to translate request and response data to and from the model required by the client. The following example shows a Zuul route filter:

**public** **class** OkHttpRoutingFilter **extends** ZuulFilter {

*@Autowired*

**private** ProxyRequestHelper helper;

*@Override*

**public** String filterType() {

**return** ROUTE\_TYPE;

}

*@Override*

**public** **int** filterOrder() {

**return** SIMPLE\_HOST\_ROUTING\_FILTER\_ORDER - 1;

}

*@Override*

**public** **boolean** shouldFilter() {

**return** RequestContext.getCurrentContext().getRouteHost() != null

&& RequestContext.getCurrentContext().sendZuulResponse();

}

*@Override*

**public** Object run() {

OkHttpClient httpClient = **new** OkHttpClient.Builder()

*// customize*

.build();

RequestContext context = RequestContext.getCurrentContext();

HttpServletRequest request = context.getRequest();

String method = request.getMethod();

String uri = **this**.helper.buildZuulRequestURI(request);

Headers.Builder headers = **new** Headers.Builder();

Enumeration<String> headerNames = request.getHeaderNames();

**while** (headerNames.hasMoreElements()) {

String name = headerNames.nextElement();

Enumeration<String> values = request.getHeaders(name);

**while** (values.hasMoreElements()) {

String value = values.nextElement();

headers.add(name, value);

}

}

InputStream inputStream = request.getInputStream();

RequestBody requestBody = null;

**if** (inputStream != null && HttpMethod.permitsRequestBody(method)) {

MediaType mediaType = null;

**if** (headers.get("Content-Type") != null) {

mediaType = MediaType.parse(headers.get("Content-Type"));

}

requestBody = RequestBody.create(mediaType, StreamUtils.copyToByteArray(inputStream));

}

Request.Builder builder = **new** Request.Builder()

.headers(headers.build())

.url(uri)

.method(method, requestBody);

Response response = httpClient.newCall(builder.build()).execute();

LinkedMultiValueMap<String, String> responseHeaders = **new** LinkedMultiValueMap<>();

**for** (Map.Entry<String, List<String>> entry : response.headers().toMultimap().entrySet()) {

responseHeaders.put(entry.getKey(), entry.getValue());

}

**this**.helper.setResponse(response.code(), response.body().byteStream(),

responseHeaders);

context.setRouteHost(null); *// prevent SimpleHostRoutingFilter from running*

**return** null;

}

}

The preceding filter translates Servlet request information into OkHttp3 request information, executes an HTTP request, and translates OkHttp3 response information to the Servlet response.

**How to Write a Post Filter**

Post filters typically manipulate the response. The following filter adds a random UUID as the X-Sample header:

**public** **class** AddResponseHeaderFilter **extends** ZuulFilter {

*@Override*

**public** String filterType() {

**return** POST\_TYPE;

}

*@Override*

**public** **int** filterOrder() {

**return** SEND\_RESPONSE\_FILTER\_ORDER - 1;

}

*@Override*

**public** **boolean** shouldFilter() {

**return** true;

}

*@Override*

**public** Object run() {

RequestContext context = RequestContext.getCurrentContext();

HttpServletResponse servletResponse = context.getResponse();

servletResponse.addHeader("X-Sample", UUID.randomUUID().toString());

**return** null;

}

}

|  |
| --- |
| [Note] |
| Other manipulations, such as transforming the response body, are much more complex and computationally intensive. |

**18.16.7 How Zuul Errors Work**

If an exception is thrown during any portion of the Zuul filter lifecycle, the error filters are executed. The SendErrorFilter is only run if RequestContext.getThrowable() is not null. It then sets specific javax.servlet.error.\* attributes in the request and forwards the request to the Spring Boot error page.

**18.16.8 Zuul Eager Application Context Loading**

Zuul internally uses Ribbon for calling the remote URLs. By default, Ribbon clients are lazily loaded by Spring Cloud on first call. This behavior can be changed for Zuul by using the following configuration, which results eager loading of the child Ribbon related Application contexts at application startup time. The following example shows how to enable eager loading:

**application.yml.**

zuul:

ribbon:

eager-load:

enabled: true

**19. Polyglot support with Sidecar**

Do you have non-JVM languages with which you want to take advantage of Eureka, Ribbon, and Config Server? The Spring Cloud Netflix Sidecar was inspired by [Netflix Prana](https://github.com/Netflix/Prana). It includes an HTTP API to get all of the instances (by host and port) for a given service. You can also proxy service calls through an embedded Zuul proxy that gets its route entries from Eureka. The Spring Cloud Config Server can be accessed directly through host lookup or through the Zuul Proxy. The non-JVM application should implement a health check so the Sidecar can report to Eureka whether the app is up or down.

To include Sidecar in your project, use the dependency with a group ID of org.springframework.cloud and artifact ID or spring-cloud-netflix-sidecar.

To enable the Sidecar, create a Spring Boot application with @EnableSidecar. This annotation includes @EnableCircuitBreaker, @EnableDiscoveryClient, and @EnableZuulProxy. Run the resulting application on the same host as the non-JVM application.

To configure the side car, add sidecar.port and sidecar.health-uri to application.yml. The sidecar.port property is the port on which the non-JVM application listens. This is so the Sidecar can properly register the application with Eureka. The sidecar.health-uri is a URI accessible on the non-JVM application that mimics a Spring Boot health indicator. It should return a JSON document that resembles the following:

**health-uri-document.**

**{**

"status":"UP"

**}**

HThe following application.yml example shows sample configuration for a Sidecar application:

**application.yml.**

server:

port: 5678

spring:

application:

name: sidecar

sidecar:

port: 8000

health-uri: http://localhost:8000/health.json

The API for the DiscoveryClient.getInstances() method is /hosts/{serviceId}. The following example response for /hosts/customers returns two instances on different hosts:

**/hosts/customers.**

**[**

**{**

"host": "myhost"**,**

"port": 9000**,**

"uri": "http://myhost:9000"**,**

"serviceId": "CUSTOMERS"**,**

"secure": **false**

**},**

**{**

"host": "myhost2"**,**

"port": 9000**,**

"uri": "http://myhost2:9000"**,**

"serviceId": "CUSTOMERS"**,**

"secure": **false**

**}**

**]**

This API is accessible to the non-JVM application (if the sidecar is on port 5678) at [http://localhost:5678/hosts/{serviceId}](http://localhost:5678/hosts/%7BserviceId%7D).

The Zuul proxy automatically adds routes for each service known in Eureka to /<serviceId>, so the customers service is available at /customers. The non-JVM application can access the customer service at <http://localhost:5678/customers> (assuming the sidecar is listening on port 5678).

If the Config Server is registered with Eureka, the non-JVM application can access it through the Zuul proxy. If the serviceId of the ConfigServer is configserverand the Sidecar is on port 5678, then it can be accessed at <http://localhost:5678/configserver>.

Non-JVM applications can take advantage of the Config Server’s ability to return YAML documents. For example, a call to <http://sidecar.local.spring.io:5678/configserver/default-master.yml> might result in a YAML document resembling the following:

eureka:

client:

serviceUrl:

defaultZone: http://localhost:8761/eureka/

password: password

info:

description: Spring Cloud Samples

url: https://github.com/spring-cloud-samples

**20. Retrying Failed Requests**

Spring Cloud Netflix offers a variety of ways to make HTTP requests. You can use a load balanced RestTemplate, Ribbon, or Feign. No matter how you choose to create your HTTP requests, there is always a chance that a request may fail. When a request fails, you may want to have the request be retried automatically. To do so when using Sping Cloud Netflix, you need to include [Spring Retry](https://github.com/spring-projects/spring-retry) on your application’s classpath. When Spring Retry is present, load-balanced RestTemplates, Feign, and Zuul automatically retry any failed requests (assuming your configuration allows doing so).

**20.1 BackOff Policies**

By default, no backoff policy is used when retrying requests. If you would like to configure a backoff policy, you need to create a bean of type LoadBalancedBackOffPolicyFactory, which is used to create a BackOffPolicy for a given service, as shown in the following example:

*@Configuration*

**public** **class** MyConfiguration {

*@Bean*

LoadBalancedBackOffPolicyFactory backOffPolicyFactory() {

**return** **new** LoadBalancedBackOffPolicyFactory() {

*@Override*

**public** BackOffPolicy createBackOffPolicy(String service) {

**return** **new** ExponentialBackOffPolicy();

}

};

}

}

**20.2 Configuration**

When you use Ribbon with Spring Retry, you can control the retry functionality by configuring certain Ribbon properties. To do so, set the client.ribbon.MaxAutoRetries, client.ribbon.MaxAutoRetriesNextServer, and client.ribbon.OkToRetryOnAllOperations properties. See the [Ribbon documentation](https://github.com/Netflix/ribbon/wiki/Getting-Started#the-properties-file-sample-clientproperties) for a description of what these properties do.

|  |
| --- |
| [Warning] |
| Enabling client.ribbon.OkToRetryOnAllOperations includes retrying POST requests, which can have an impact on the server’s resources, due to the buffering of the request body. |

In addition, you may want to retry requests when certain status codes are returned in the response. You can list the response codes you would like the Ribbon client to retry by setting the clientName.ribbon.retryableStatusCodes property, as shown in the following example:

clientName:

ribbon:

retryableStatusCodes: 404,502

You can also create a bean of type LoadBalancedRetryPolicy and implement the retryableStatusCode method to retry a request given the status code.

**20.2.1 Zuul**

You can turn off Zuul’s retry functionality by setting zuul.retryable to false. You can also disable retry functionality on a route-by-route basis by setting zuul.routes.routename.retryable to false.

**21. HTTP Clients**

Spring Cloud Netflix automatically creates the HTTP client used by Ribbon, Feign, and Zuul for you. However, you can also provide your own HTTP clients customized as you need them to be. To do so, you can create a bean of type ClosableHttpClient if you are using the Apache Http Cient or OkHttpClient if you are using OK HTTP.

|  |
| --- |
| [Note] |
| When you create your own HTTP client, you are also responsible for implementing the correct connection management strategies for these clients. Doing so improperly can result in resource management issues. |

**Part IV. Spring Cloud OpenFeign**

**Finchley.SR1**

This project provides OpenFeign integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms.

**22. Declarative REST Client: Feign**

[Feign](https://github.com/Netflix/feign) is a declarative web service client. It makes writing web service clients easier. To use Feign create an interface and annotate it. It has pluggable annotation support including Feign annotations and JAX-RS annotations. Feign also supports pluggable encoders and decoders. Spring Cloud adds support for Spring MVC annotations and for using the same HttpMessageConverters used by default in Spring Web. Spring Cloud integrates Ribbon and Eureka to provide a load balanced http client when using Feign.

**22.1 How to Include Feign**

To include Feign in your project use the starter with group org.springframework.cloud and artifact id spring-cloud-starter-openfeign. See the [Spring Cloud Project page](https://projects.spring.io/spring-cloud/) for details on setting up your build system with the current Spring Cloud Release Train.

Example spring boot app

*@SpringBootApplication*

*@EnableFeignClients*

**public** **class** Application {

**public** **static** **void** main(String[] args) {

SpringApplication.run(Application.**class**, args);

}

}

**StoreClient.java.**

*@FeignClient("stores")*

**public** **interface** StoreClient {

*@RequestMapping(method = RequestMethod.GET, value = "/stores")*

List<Store> getStores();

*@RequestMapping(method = RequestMethod.POST, value = "/stores/{storeId}", consumes = "application/json")*

Store update(*@PathVariable("storeId")* Long storeId, Store store);

}

In the @FeignClient annotation the String value ("stores" above) is an arbitrary client name, which is used to create a Ribbon load balancer (see [below for details of Ribbon support](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-ribbon)). You can also specify a URL using the url attribute (absolute value or just a hostname). The name of the bean in the application context is the fully qualified name of the interface. To specify your own alias value you can use the qualifier value of the @FeignClient annotation.

The Ribbon client above will want to discover the physical addresses for the "stores" service. If your application is a Eureka client then it will resolve the service in the Eureka service registry. If you don’t want to use Eureka, you can simply configure a list of servers in your external configuration (see [above for example](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-ribbon-without-eureka)).

**22.2 Overriding Feign Defaults**

A central concept in Spring Cloud’s Feign support is that of the named client. Each feign client is part of an ensemble of components that work together to contact a remote server on demand, and the ensemble has a name that you give it as an application developer using the @FeignClient annotation. Spring Cloud creates a new ensemble as an ApplicationContext on demand for each named client using FeignClientsConfiguration. This contains (amongst other things) an feign.Decoder, a feign.Encoder, and a feign.Contract.

Spring Cloud lets you take full control of the feign client by declaring additional configuration (on top of the FeignClientsConfiguration) using @FeignClient. Example:

*@FeignClient(name = "stores", configuration = FooConfiguration.class)*

**public** **interface** StoreClient {

*//..*

}

In this case the client is composed from the components already in FeignClientsConfiguration together with any in FooConfiguration (where the latter will override the former).

|  |
| --- |
| [Note] |
| FooConfiguration does not need to be annotated with @Configuration. However, if it is, then take care to exclude it from any @ComponentScan that would otherwise include this configuration as it will become the default source for feign.Decoder, feign.Encoder, feign.Contract, etc., when specified. This can be avoided by putting it in a separate, non-overlapping package from any @ComponentScan or @SpringBootApplication, or it can be explicitly excluded in @ComponentScan. | |
| [Note] | |
| The serviceId attribute is now deprecated in favor of the name attribute. | |

|  |
| --- |
| [Warning] |
| Previously, using the url attribute, did not require the name attribute. Using name is now required. |

Placeholders are supported in the name and url attributes.

*@FeignClient(name = "${feign.name}", url = "${feign.url}")*

**public** **interface** StoreClient {

*//..*

}

Spring Cloud Netflix provides the following beans by default for feign (BeanType beanName: ClassName):

* Decoder feignDecoder: ResponseEntityDecoder (which wraps a SpringDecoder)
* Encoder feignEncoder: SpringEncoder
* Logger feignLogger: Slf4jLogger
* Contract feignContract: SpringMvcContract
* Feign.Builder feignBuilder: HystrixFeign.Builder
* Client feignClient: if Ribbon is enabled it is a LoadBalancerFeignClient, otherwise the default feign client is used.

The OkHttpClient and ApacheHttpClient feign clients can be used by setting feign.okhttp.enabled or feign.httpclient.enabled to true, respectively, and having them on the classpath. You can customize the HTTP client used by providing a bean of either ClosableHttpClient when using Apache or OkHttpClientwhen using OK HTTP.

Spring Cloud Netflix *does not* provide the following beans by default for feign, but still looks up beans of these types from the application context to create the feign client:

* Logger.Level
* Retryer
* ErrorDecoder
* Request.Options
* Collection<RequestInterceptor>
* SetterFactory

Creating a bean of one of those type and placing it in a @FeignClient configuration (such as FooConfiguration above) allows you to override each one of the beans described. Example:

*@Configuration*

**public** **class** FooConfiguration {

*@Bean*

**public** Contract feignContract() {

**return** **new** feign.Contract.Default();

}

*@Bean*

**public** BasicAuthRequestInterceptor basicAuthRequestInterceptor() {

**return** **new** BasicAuthRequestInterceptor("user", "password");

}

}

This replaces the SpringMvcContract with feign.Contract.Default and adds a RequestInterceptor to the collection of RequestInterceptor.

@FeignClient also can be configured using configuration properties.

application.yml

feign:

client:

config:

feignName:

connectTimeout: 5000

readTimeout: 5000

loggerLevel: full

errorDecoder: com.example.SimpleErrorDecoder

retryer: com.example.SimpleRetryer

requestInterceptors:

- com.example.FooRequestInterceptor

- com.example.BarRequestInterceptor

decode404: **false**

encoder: com.example.SimpleEncoder

decoder: com.example.SimpleDecoder

contract: com.example.SimpleContract

Default configurations can be specified in the @EnableFeignClients attribute defaultConfiguration in a similar manner as described above. The difference is that this configuration will apply to *all* feign clients.

If you prefer using configuration properties to configured all @FeignClient, you can create configuration properties with default feign name.

application.yml

feign:

client:

config:

default:

connectTimeout: 5000

readTimeout: 5000

loggerLevel: basic

If we create both @Configuration bean and configuration properties, configuration properties will win. It will override @Configuration values. But if you want to change the priority to @Configuration, you can change feign.client.default-to-properties to false.

|  |
| --- |
| [Note] |
| If you need to use ThreadLocal bound variables in your RequestInterceptor`s you will need to either set the thread isolation strategy for Hystrix to `SEMAPHORE or disable Hystrix in Feign. |

application.yml

*# To disable Hystrix in Feign*

feign:

hystrix:

enabled: **false**

*# To set thread isolation to SEMAPHORE*

hystrix:

command:

default:

execution:

isolation:

strategy: SEMAPHORE

**22.3 Creating Feign Clients Manually**

In some cases it might be necessary to customize your Feign Clients in a way that is not possible using the methods above. In this case you can create Clients using the[Feign Builder API](https://github.com/OpenFeign/feign/#basics). Below is an example which creates two Feign Clients with the same interface but configures each one with a separate request interceptor.

*@Import(FeignClientsConfiguration.class)*

**class** FooController {

**private** FooClient fooClient;

**private** FooClient adminClient;

*@Autowired*

**public** FooController(Decoder decoder, Encoder encoder, Client client, Contract contract) {

**this**.fooClient = Feign.builder().client(client)

.encoder(encoder)

.decoder(decoder)

.contract(contract)

.requestInterceptor(**new** BasicAuthRequestInterceptor("user", "user"))

.target(FooClient.**class**, "http://PROD-SVC");

**this**.adminClient = Feign.builder().client(client)

.encoder(encoder)

.decoder(decoder)

.contract(contract)

.requestInterceptor(**new** BasicAuthRequestInterceptor("admin", "admin"))

.target(FooClient.**class**, "http://PROD-SVC");

}

}

|  |
| --- |
| [Note] |
| In the above example FeignClientsConfiguration.class is the default configuration provided by Spring Cloud Netflix. |
| [Note] |
| PROD-SVC is the name of the service the Clients will be making requests to. | |

|  |
| --- |
| [Note] |
| The Feign Contract object defines what annotations and values are valid on interfaces. The autowired Contract bean provides supports for SpringMVC annotations, instead of the default Feign native annotations. |

**22.4 Feign Hystrix Support**

If Hystrix is on the classpath and feign.hystrix.enabled=true, Feign will wrap all methods with a circuit breaker. Returning a com.netflix.hystrix.HystrixCommand is also available. This lets you use reactive patterns (with a call to .toObservable() or .observe() or asynchronous use (with a call to .queue()).

To disable Hystrix support on a per-client basis create a vanilla Feign.Builder with the "prototype" scope, e.g.:

*@Configuration*

**public** **class** FooConfiguration {

*@Bean*

*@Scope("prototype")*

**public** Feign.Builder feignBuilder() {

**return** Feign.builder();

}

}

|  |
| --- |
| [Warning] |
| Prior to the Spring Cloud Dalston release, if Hystrix was on the classpath Feign would have wrapped all methods in a circuit breaker by default. This default behavior was changed in Spring Cloud Dalston in favor for an opt-in approach. |

**22.5 Feign Hystrix Fallbacks**

Hystrix supports the notion of a fallback: a default code path that is executed when they circuit is open or there is an error. To enable fallbacks for a given @FeignClientset the fallback attribute to the class name that implements the fallback. You also need to declare your implementation as a Spring bean.

*@FeignClient(name = "hello", fallback = HystrixClientFallback.class)*

**protected** **interface** HystrixClient {

*@RequestMapping(method = RequestMethod.GET, value = "/hello")*

Hello iFailSometimes();

}

**static** **class** HystrixClientFallback **implements** HystrixClient {

*@Override*

**public** Hello iFailSometimes() {

**return** **new** Hello("fallback");

}

}

If one needs access to the cause that made the fallback trigger, one can use the fallbackFactory attribute inside @FeignClient.

*@FeignClient(name = "hello", fallbackFactory = HystrixClientFallbackFactory.class)*

**protected** **interface** HystrixClient {

*@RequestMapping(method = RequestMethod.GET, value = "/hello")*

Hello iFailSometimes();

}

*@Component*

**static** **class** HystrixClientFallbackFactory **implements** FallbackFactory<HystrixClient> {

*@Override*

**public** HystrixClient create(Throwable cause) {

**return** **new** HystrixClient() {

*@Override*

**public** Hello iFailSometimes() {

**return** **new** Hello("fallback; reason was: " + cause.getMessage());

}

};

}

}

|  |
| --- |
| [Warning] |
| There is a limitation with the implementation of fallbacks in Feign and how Hystrix fallbacks work. Fallbacks are currently not supported for methods that return com.netflix.hystrix.HystrixCommand and rx.Observable. |

**22.6 Feign and @Primary**

When using Feign with Hystrix fallbacks, there are multiple beans in the ApplicationContext of the same type. This will cause @Autowired to not work because there isn’t exactly one bean, or one marked as primary. To work around this, Spring Cloud Netflix marks all Feign instances as @Primary, so Spring Framework will know which bean to inject. In some cases, this may not be desirable. To turn off this behavior set the primary attribute of @FeignClient to false.

*@FeignClient(name = "hello", primary = false)*

**public** **interface** HelloClient {

*// methods here*

}

**22.7 Feign Inheritance Support**

Feign supports boilerplate apis via single-inheritance interfaces. This allows grouping common operations into convenient base interfaces.

**UserService.java.**

**public** **interface** UserService {

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

User getUser(*@PathVariable("id")* **long** id);

}

**UserResource.java.**

*@RestController*

**public** **class** UserResource **implements** UserService {

}

**UserClient.java.**

**package** project.user;

*@FeignClient("users")*

**public** **interface** UserClient **extends** UserService {

}

|  |
| --- |
| [Note] |
| It is generally not advisable to share an interface between a server and a client. It introduces tight coupling, and also actually doesn’t work with Spring MVC in its current form (method parameter mapping is not inherited). |

**22.8 Feign request/response compression**

You may consider enabling the request or response GZIP compression for your Feign requests. You can do this by enabling one of the properties:

feign.compression.request.enabled=true

feign.compression.response.enabled=true

Feign request compression gives you settings similar to what you may set for your web server:

feign.compression.request.enabled=true

feign.compression.request.mime-types=text/xml,application/xml,application/json

feign.compression.request.min-request-size=2048

These properties allow you to be selective about the compressed media types and minimum request threshold length.

**22.9 Feign logging**

A logger is created for each Feign client created. By default the name of the logger is the full class name of the interface used to create the Feign client. Feign logging only responds to the DEBUG level.

**application.yml.**

logging.level.project.user.UserClient: DEBUG

The Logger.Level object that you may configure per client, tells Feign how much to log. Choices are:

* NONE, No logging (**DEFAULT**).
* BASIC, Log only the request method and URL and the response status code and execution time.
* HEADERS, Log the basic information along with request and response headers.
* FULL, Log the headers, body, and metadata for both requests and responses.

For example, the following would set the Logger.Level to FULL:

*@Configuration*

**public** **class** FooConfiguration {

*@Bean*

Logger.Level feignLoggerLevel() {

**return** Logger.Level.FULL;

}

}

OtherClass.someMethod(myprop.get());

}

}

stripped). The proxy uses Ribbon to locate an instance to forward to

via discovery, and all requests are executed in a

<<hystrix-fallbacks-for-routes, hystrix command>>, so

failures will show up in Hystrix metrics, and once the circuit is open

the proxy will not try to contact the service.

**Part V. Spring Cloud Stream**

**23. Quick Start**

You can try Spring Cloud Stream in less then 5 min even before you jump into any details by following this three-step guide.

We show you how to create a Spring Cloud Stream application that receives messages coming from the messaging middleware of your choice (more on this later) and logs received messages to the console. We call it LoggingConsumer. While not very practical, it provides a good introduction to some of the main concepts and abstractions, making it easier to digest the rest of this user guide.

The three steps are as follows:

1. [Section 23.1, “Creating a Sample Application by Using Spring Initializr”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-preface-creating-sample-application)
2. [Section 23.2, “Importing the Project into Your IDE”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-preface-importing-project)
3. [Section 23.3, “Adding a Message Handler, Building, and Running”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-preface-adding-message-handler)

**23.1 Creating a Sample Application by Using Spring Initializr**

To get started, visit the [Spring Initializr](https://start.spring.io/). From there, you can generate our LoggingConsumer application. To do so:

1. In the **Dependencies** section, start typing stream. When the “Cloud Stream” option should appears, select it.
2. Start typing either 'kafka' or 'rabbit'.
3. Select “Kafka” or “RabbitMQ”.

Basically, you choose the messaging middleware to which your application binds. We recommend using the one you have already installed or feel more comfortable with installing and running. Also, as you can see from the Initilaizer screen, there are a few other options you can choose. For example, you can choose Gradle as your build tool instead of Maven (the default).

1. In the **Artifact** field, type 'logging-consumer'.

The value of the **Artifact** field becomes the application name. If you chose RabbitMQ for the middleware, your Spring Initializr should now be as follows:

1. Click the **Generate Project** button.

Doing so downloads the zipped version of the generated project to your hard drive.

1. Unzip the file into the folder you want to use as your project directory.

|  |
| --- |
| [Tip] |
| We encourage you to explore the many possibilities available in the Spring Initializr. It lets you create many different kinds of Spring applications. |

**23.2 Importing the Project into Your IDE**

Now you can import the project into your IDE. Keep in mind that, depending on the IDE, you may need to follow a specific import procedure. For example, depending on how the project was generated (Maven or Gradle), you may need to follow specific import procedure (for example, in Eclipse or STS, you need to use File → Import → Maven → Existing Maven Project).

Once imported, the project must have no errors of any kind. Also, src/main/java should contain com.example.loggingconsumer.LoggingConsumerApplication.

Technically, at this point, you can run the application’s main class. It is already a valid Spring Boot application. However, it does not do anything, so we want to add some code.

**23.3 Adding a Message Handler, Building, and Running**

Modify the com.example.loggingconsumer.LoggingConsumerApplication class to look as follows:

*@SpringBootApplication*

*@EnableBinding(Sink.class)*

**public** **class** LoggingConsumerApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.run(LoggingConsumerApplication.**class**, args);

}

*@StreamListener(Sink.INPUT)*

**public** **void** handle(Person person) {

System.out.println("Received: " + person);

}

**public** **static** **class** Person {

**private** String name;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String toString() {

**return** **this**.name;

}

}

}

As you can see from the preceding listing:

* We have enabled Sink binding (input-no-output) by using @EnableBinding(Sink.class). Doing so signals to the framework to initiate binding to the messaging middleware, where it automatically creates the destination (that is, queue, topic, and others) that are bound to the Sink.INPUT channel.
* We have added a handler method to receive incoming messages of type Person. Doing so lets you see one of the core features of the framework: It tries to automatically convert incoming message payloads to type Person.

You now have a fully functional Spring Cloud Stream application that does listens for messages. From here, for simplicity, we assume you selected RabbitMQ in [step one](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-preface-creating-sample-application). Assuming you have RabbitMQ installed and running, you can start the application by running its main method in your IDE.

You should see following output:

--- [ main] c.s.b.r.p.RabbitExchangeQueueProvisioner : declaring queue for inbound: input.anonymous.CbMIwdkJSBO1ZoPDOtHtCg, bound to: input

--- [ main] o.s.a.r.c.CachingConnectionFactory : Attempting to connect to: [localhost:5672]

--- [ main] o.s.a.r.c.CachingConnectionFactory : Created new connection: rabbitConnectionFactory#2a3a299:0/SimpleConnection@66c83fc8. . .

. . .

--- [ main] o.s.i.a.i.AmqpInboundChannelAdapter : started inbound.input.anonymous.CbMIwdkJSBO1ZoPDOtHtCg

. . .

--- [ main] c.e.l.LoggingConsumerApplication : Started LoggingConsumerApplication in 2.531 seconds (JVM running for 2.897)

Go to the RabbitMQ management console or any other RabbitMQ client and send a message to input.anonymous.CbMIwdkJSBO1ZoPDOtHtCg. The anonymous.CbMIwdkJSBO1ZoPDOtHtCg part represents the group name and is generated, so it is bound to be different in your environment. For something more predictable, you can use an explicit group name by setting spring.cloud.stream.bindings.input.group=hello (or whatever name you like).

The contents of the message should be a JSON representation of the Person class, as follows:

{"name":"Sam Spade"}

Then, in your console, you should see:

Received: Sam Spade

You can also build and package your application into a boot jar (by using ./mvnw clean install) and run the built JAR by using the java -jar command.

Now you have a working (albeit very basic) Spring Cloud Stream application.

**24. What’s New in 2.0?**

Spring Cloud Stream introduces a number of new features, enhancements, and changes. The following sections outline the most notable ones:

* [Section 24.1, “New Features and Components”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-preface-new-features)
* [Section 24.2, “Notable Enhancements”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-preface-notable-enhancements)

**24.1 New Features and Components**

* **Polling Consumers**: Introduction of polled consumers, which lets the application control message processing rates. See “[Section 27.3.4, “Using Polled Consumers”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-streams-overview-using-polled-consumers)” for more details. You can also read [this blog post](https://spring.io/blog/2018/02/27/spring-cloud-stream-2-0-polled-consumers) for more details.
* **Micrometer Support**: Metrics has been switched to use [Micrometer](https://micrometer.io/). MeterRegistry is also provided as a bean so that custom applications can autowire it to capture custom metrics. See “[Chapter 35, *Metrics Emitter*](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-metrics-emitter)” for more details.
* **New Actuator Binding Controls**: New actuator binding controls let you both visualize and control the Bindings lifecycle. For more details, see [Section 28.6, “Binding visualization and control”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_binding_visualization_and_control).
* **Configurable RetryTemplate**: Aside from providing properties to configure RetryTemplate, we now let you provide your own template, effectively overriding the one provided by the framework. To use it, configure it as a @Bean in your application.

**24.2 Notable Enhancements**

This version includes the following notable enhancements:

* [Section 24.2.1, “Both Actuator and Web Dependencies Are Now Optional”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-preface-actuator-web-dependencies)
* [Section 24.2.2, “Content-type Negotiation Improvements”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-preface-content-type-negotiation-improvements)
* [Section 24.3, “Notable Deprecations”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-preface-notable-deprecations)

**24.2.1 Both Actuator and Web Dependencies Are Now Optional**

This change slims down the footprint of the deployed application in the event neither actuator nor web dependencies required. It also lets you switch between the reactive and conventional web paradigms by manually adding one of the following dependencies.

The following listing shows how to add the conventional web framework:

<dependency>

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

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

</dependency>

The following listing shows how to add the reactive web framework:

<dependency>

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

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

</dependency>

The following list shows how to add the actuator dependency:

<dependency>

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

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

</dependency>

**24.2.2 Content-type Negotiation Improvements**

One of the core themes for verion 2.0 is improvements (in both consistency and performance) around content-type negotiation and message conversion. The following summary outlines the notable changes and improvements in this area. See the “[Chapter 30, *Content Type Negotiation*](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#content-type-management)” section for more details. Also [this blog post](https://spring.io/blog/2018/02/26/spring-cloud-stream-2-0-content-type-negotiation-and-transformation)contains more detail.

* All message conversion is now handled **only** by MessageConverter objects.
* We introduced the @StreamMessageConverter annotation to provide custom MessageConverter objects.
* We introduced the default Content Type as application/json, which needs to be taken into consideration when migrating 1.3 application or operating in the mixed mode (that is, 1.3 producer → 2.0 consumer).
* Messages with textual payloads and a contentType of text/…​ or …​/json are no longer converted to Message<String> for cases where the argument type of the provided MessageHandler can not be determined (that is, public void handle(Message<?> message) or public void handle(Object payload)). Furthermore, a strong argument type may not be enough to properly convert messages, so the contentType header may be used as a supplement by some MessageConverters.

**24.3 Notable Deprecations**

As of version 2.0, the following items have been deprecated:

* [Section 24.3.1, “Java Serialization (Java Native and Kryo)”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-preface-deprecation-java-serialization)
* [Section 24.3.2, “Deprecated Classes and Methods”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-preface-deprecation-classes-methods)

**24.3.1 Java Serialization (Java Native and Kryo)**

JavaSerializationMessageConverter and KryoMessageConverter remain for now. However, we plan to move them out of the core packages and support in the future. The main reason for this deprecation is to flag the issue that type-based, language-specific serialization could cause in distributed environments, where Producers and Consumers may depend on different JVM versions or have different versions of supporting libraries (that is, Kryo). We also wanted to draw the attention to the fact that Consumers and Producers may not even be Java-based, so polyglot style serialization (i.e., JSON) is better suited.

**24.3.2 Deprecated Classes and Methods**

The following is a quick summary of notable deprecations. See the corresponding {spring-cloud-stream-javadoc-current}[javadoc] for more details.

* SharedChannelRegistry. Use SharedBindingTargetRegistry.
* Bindings. Beans qualified by it are already uniquely identified by their type — for example, provided Source, Processor, or custom bindings:

public interface Sample {

String OUTPUT = "sampleOutput";

@Output(Sample.OUTPUT)

MessageChannel output();

}

* HeaderMode.raw. Use none, headers or embeddedHeaders
* ProducerProperties.partitionKeyExtractorClass in favor of partitionKeyExtractorName and ProducerProperties.partitionSelectorClass in favor of partitionSelectorName. This change ensures that both components are Spring configured and managed and are referenced in a Spring-friendly way.
* BinderAwareRouterBeanPostProcessor. While the component remains, it is no longer a BeanPostProcessor and will be renamed in the future.
* BinderProperties.setEnvironment(Properties environment). Use BinderProperties.setEnvironment(Map<String, Object> environment).

This section goes into more detail about how you can work with Spring Cloud Stream. It covers topics such as creating and running stream applications.

**25. Introducing Spring Cloud Stream**

Spring Cloud Stream is a framework for building message-driven microservice applications. Spring Cloud Stream builds upon Spring Boot to create standalone, production-grade Spring applications and uses Spring Integration to provide connectivity to message brokers. It provides opinionated configuration of middleware from several vendors, introducing the concepts of persistent publish-subscribe semantics, consumer groups, and partitions.

You can add the @EnableBinding annotation to your application to get immediate connectivity to a message broker, and you can add @StreamListener to a method to cause it to receive events for stream processing. The following example shows a sink application that receives external messages:

*@SpringBootApplication*

*@EnableBinding(Sink.class)*

**public** **class** VoteRecordingSinkApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.run(VoteRecordingSinkApplication.**class**, args);

}

*@StreamListener(Sink.INPUT)*

**public** **void** processVote(Vote vote) {

votingService.recordVote(vote);

}

}

The @EnableBinding annotation takes one or more interfaces as parameters (in this case, the parameter is a single Sink interface). An interface declares input and output channels. Spring Cloud Stream provides the Source, Sink, and Processor interfaces. You can also define your own interfaces.

The following listing shows the definition of the Sink interface:

**public** **interface** Sink {

String INPUT = "input";

*@Input(Sink.INPUT)*

SubscribableChannel input();

}

The @Input annotation identifies an input channel, through which received messages enter the application. The @Output annotation identifies an output channel, through which published messages leave the application. The @Input and @Output annotations can take a channel name as a parameter. If a name is not provided, the name of the annotated method is used.

Spring Cloud Stream creates an implementation of the interface for you. You can use this in the application by autowiring it, as shown in the following example (from a test case):

*@RunWith(SpringJUnit4ClassRunner.class)*

*@SpringApplicationConfiguration(classes = VoteRecordingSinkApplication.class)*

*@WebAppConfiguration*

*@DirtiesContext*

**public** **class** StreamApplicationTests {

*@Autowired*

**private** Sink sink;

*@Test*

**public** **void** contextLoads() {

assertNotNull(**this**.sink.input());

}

}

**26. Main Concepts**

Spring Cloud Stream provides a number of abstractions and primitives that simplify the writing of message-driven microservice applications. This section gives an overview of the following:

* [Spring Cloud Stream’s application model](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-application-model)
* [Section 26.2, “The Binder Abstraction”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-binder-abstraction)
* [Persistent publish-subscribe support](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-persistent-publish-subscribe-support)
* [Consumer group support](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#consumer-groups)
* [Partitioning support](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#partitioning)
* [A pluggable Binder SPI](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-binder-api)

**26.1 Application Model**

A Spring Cloud Stream application consists of a middleware-neutral core. The application communicates with the outside world through input and output channels injected into it by Spring Cloud Stream. Channels are connected to external brokers through middleware-specific Binder implementations.

**Figure 26.1. Spring Cloud Stream Application**

**26.1.1 Fat JAR**

Spring Cloud Stream applications can be run in stand-alone mode from your IDE for testing. To run a Spring Cloud Stream application in production, you can create an executable (or “fat”) JAR by using the standard Spring Boot tooling provided for Maven or Gradle. See the [Spring Boot Reference Guide](https://docs.spring.io/spring-boot/docs/current/reference/html/howto-build.html#howto-create-an-executable-jar-with-maven) for more details.

**26.2 The Binder Abstraction**

Spring Cloud Stream provides Binder implementations for [Kafka](https://github.com/spring-cloud/spring-cloud-stream/tree/master/spring-cloud-stream-binders/spring-cloud-stream-binder-kafka) and [Rabbit MQ](https://github.com/spring-cloud/spring-cloud-stream/tree/master/spring-cloud-stream-binders/spring-cloud-stream-binder-rabbit). Spring Cloud Stream also includes a [TestSupportBinder](https://github.com/spring-cloud/spring-cloud-stream/blob/master/spring-cloud-stream-test-support/src/main/java/org/springframework/cloud/stream/test/binder/TestSupportBinder.java), which leaves a channel unmodified so that tests can interact with channels directly and reliably assert on what is received. You can also use the extensible API to write your own Binder.

Spring Cloud Stream uses Spring Boot for configuration, and the Binder abstraction makes it possible for a Spring Cloud Stream application to be flexible in how it connects to middleware. For example, deployers can dynamically choose, at runtime, the destinations (such as the Kafka topics or RabbitMQ exchanges) to which channels connect. Such configuration can be provided through external configuration properties and in any form supported by Spring Boot (including application arguments, environment variables, and application.yml or application.properties files). In the sink example from the [Chapter 25, *Introducing Spring Cloud Stream*](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-introducing) section, setting the spring.cloud.stream.bindings.input.destination application property to raw-sensor-data causes it to read from the raw-sensor-data Kafka topic or from a queue bound to the raw-sensor-data RabbitMQ exchange.

Spring Cloud Stream automatically detects and uses a binder found on the classpath. You can use different types of middleware with the same code. To do so, include a different binder at build time. For more complex use cases, you can also package multiple binders with your application and have it choose the binder( and even whether to use different binders for different channels) at runtime.

**26.3 Persistent Publish-Subscribe Support**

Communication between applications follows a publish-subscribe model, where data is broadcast through shared topics. This can be seen in the following figure, which shows a typical deployment for a set of interacting Spring Cloud Stream applications.

**Figure 26.2. Spring Cloud Stream Publish-Subscribe**

Data reported by sensors to an HTTP endpoint is sent to a common destination named raw-sensor-data. From the destination, it is independently processed by a microservice application that computes time-windowed averages and by another microservice application that ingests the raw data into HDFS (Hadoop Distributed File System). In order to process the data, both applications declare the topic as their input at runtime.

The publish-subscribe communication model reduces the complexity of both the producer and the consumer and lets new applications be added to the topology without disruption of the existing flow. For example, downstream from the average-calculating application, you can add an application that calculates the highest temperature values for display and monitoring. You can then add another application that interprets the same flow of averages for fault detection. Doing all communication through shared topics rather than point-to-point queues reduces coupling between microservices.

While the concept of publish-subscribe messaging is not new, Spring Cloud Stream takes the extra step of making it an opinionated choice for its application model. By using native middleware support, Spring Cloud Stream also simplifies use of the publish-subscribe model across different platforms.

**26.4 Consumer Groups**

While the publish-subscribe model makes it easy to connect applications through shared topics, the ability to scale up by creating multiple instances of a given application is equally important. When doing so, different instances of an application are placed in a competing consumer relationship, where only one of the instances is expected to handle a given message.

Spring Cloud Stream models this behavior through the concept of a consumer group. (Spring Cloud Stream consumer groups are similar to and inspired by Kafka consumer groups.) Each consumer binding can use the spring.cloud.stream.bindings.<channelName>.group property to specify a group name. For the consumers shown in the following figure, this property would be set as spring.cloud.stream.bindings.<channelName>.group=hdfsWrite or spring.cloud.stream.bindings.<channelName>.group=average.

**Figure 26.3. Spring Cloud Stream Consumer Groups**

All groups that subscribe to a given destination receive a copy of published data, but only one member of each group receives a given message from that destination. By default, when a group is not specified, Spring Cloud Stream assigns the application to an anonymous and independent single-member consumer group that is in a publish-subscribe relationship with all other consumer groups.

**26.5 Consumer Types**

Two types of consumer are supported:

* Message-driven (sometimes referred to as Asynchronous)
* Polled (sometimes referred to as Synchronous)

Prior to version 2.0, only asynchronous consumers were supported. A message is delivered as soon as it is available and a thread is available to process it.

When you wish to control the rate at which messages are processed, you might want to use a synchronous consumer.

**26.5.1 Durability**

Consistent with the opinionated application model of Spring Cloud Stream, consumer group subscriptions are durable. That is, a binder implementation ensures that group subscriptions are persistent and that, once at least one subscription for a group has been created, the group receives messages, even if they are sent while all applications in the group are stopped.

|  |
| --- |
| [Note] |
| Anonymous subscriptions are non-durable by nature. For some binder implementations (such as RabbitMQ), it is possible to have non-durable group subscriptions. |

In general, it is preferable to always specify a consumer group when binding an application to a given destination. When scaling up a Spring Cloud Stream application, you must specify a consumer group for each of its input bindings. Doing so prevents the application’s instances from receiving duplicate messages (unless that behavior is desired, which is unusual).

**26.6 Partitioning Support**

Spring Cloud Stream provides support for partitioning data between multiple instances of a given application. In a partitioned scenario, the physical communication medium (such as the broker topic) is viewed as being structured into multiple partitions. One or more producer application instances send data to multiple consumer application instances and ensure that data identified by common characteristics are processed by the same consumer instance.

Spring Cloud Stream provides a common abstraction for implementing partitioned processing use cases in a uniform fashion. Partitioning can thus be used whether the broker itself is naturally partitioned (for example, Kafka) or not (for example, RabbitMQ).

**Figure 26.4. Spring Cloud Stream Partitioning**

Partitioning is a critical concept in stateful processing, where it is critical (for either performance or consistency reasons) to ensure that all related data is processed together. For example, in the time-windowed average calculation example, it is important that all measurements from any given sensor are processed by the same application instance.

|  |
| --- |
| [Note] |
| To set up a partitioned processing scenario, you must configure both the data-producing and the data-consuming ends. |

**27. Programming Model**

To understand the programming model, you should be familiar with the following core concepts:

* **Destination Binders:** Components responsible to provide integration with the external messaging systems.
* **Destination Bindings:** Bridge between the external messaging systems and application provided *Producers* and *Consumers* of messages (created by the Destination Binders).
* **Message:** The canonical data structure used by producers and consumers to communicate with Destination Binders (and thus other applications via external messaging systems).

**27.1 Destination Binders**

Destination Binders are extension components of Spring Cloud Stream responsible for providing the necessary configuration and implementation to facilitate integration with external messaging systems. This integration is responsible for connectivity, delegation, and routing of messages to and from producers and consumers, data type conversion, invocation of the user code, and more.

Binders handle a lot of the boiler plate responsibilities that would otherwise fall on your shoulders. However, to accomplish that, the binder still needs some help in the form of minimalistic yet required set of instructions from the user, which typically come in the form of some type of configuration.

While it is out of scope of this section to discuss all of the available binder and binding configuration options (the rest of the manual covers them extensively), *Destination Binding* does require special attention. The next section discusses it in detail.

**27.2 Destination Bindings**

As stated earlier, *Destination Bindings* provide a bridge between the external messaging system and application-provided *Producers* and *Consumers*.

Applying the @EnableBinding annotation to one of the application’s configuration classes defines a destination binding. The @EnableBinding annotation itself is meta-annotated with @Configuration and triggers the configuration of the Spring Cloud Stream infrastructure.

The following example shows a fully configured and functioning Spring Cloud Stream application that receives the payload of the message from the INPUT destination as a String type (see [Chapter 30, *Content Type Negotiation*](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#content-type-management) section), logs it to the console and sends it to the OUTPUT destination after converting it to upper case.

*@SpringBootApplication*

*@EnableBinding(Processor.class)*

**public** **class** MyApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.run(MyApplication.**class**, args);

}

*@StreamListener(Processor.INPUT)*

*@SendTo(Processor.OUTPUT)*

**public** String handle(String value) {

System.out.println("Received: " + value);

**return** value.toUpperCase();

}

}

As you can see the @EnableBinding annotation can take one or more interface classes as parameters. The parameters are referred to as *bindings*, and they contain methods representing *bindable components*. These components are typically message channels (see [Spring Messaging](https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-messaging.html)) for channel-based binders (such as Rabbit, Kafka, and others). However other types of bindings can provide support for the native features of the corresponding technology. For example Kafka Streams binder (formerly known as KStream) allows native bindings directly to Kafka Streams (see [Kafka Streams](https://docs.spring.io/autorepo/docs/spring-cloud-stream-binder-kafka-docs/1.1.0.M1/reference/htmlsingle/) for more details).

Spring Cloud Stream already provides *binding* interfaces for typical message exchange contracts, which include:

* **Sink:** Identifies the contract for the message consumer by providing the destination from which the message is consumed.
* **Source:** Identifies the contract for the message producer by providing the destination to which the produced message is sent.
* **Processor:** Encapsulates both the sink and the source contracts by exposing two destinations that allow consumption and production of messages.

**public** **interface** Sink {

String INPUT = "input";

*@Input(Sink.INPUT)*

SubscribableChannel input();

}

**public** **interface** Source {

String OUTPUT = "output";

*@Output(Source.OUTPUT)*

MessageChannel output();

}

**public** **interface** Processor **extends** Source, Sink {}

While the preceding example satisfies the majority of cases, you can also define your own contracts by defining your own bindings interfaces and use @Input and @Output annotations to identify the actual *bindable components*.

For example:

**public** **interface** Barista {

*@Input*

SubscribableChannel orders();

*@Output*

MessageChannel hotDrinks();

*@Output*

MessageChannel coldDrinks();

}

Using the interface shown in the preceding example as a parameter to @EnableBinding triggers the creation of the three bound channels named orders, hotDrinks, and coldDrinks, respectively.

You can provide as many binding interfaces as you need, as arguments to the @EnableBinding annotation, as shown in the following example:

@EnableBinding(value = { Orders.**class**, Payment.**class** })

In Spring Cloud Stream, the bindable MessageChannel components are the Spring Messaging MessageChannel (for outbound) and its extension, SubscribableChannel, (for inbound).

**Pollable Destination Binding**

While the previously described bindings support event-based message consumption, sometimes you need more control, such as rate of consumption.

Starting with version 2.0, you can now bind a pollable consumer:

The following example shows how to bind a pollable consumer:

**public** **interface** PolledBarista {

*@Input*

PollableMessageSource orders();

. . .

}

In this case, an implementation of PollableMessageSource is bound to the orders “channel”. See [Section 27.3.4, “Using Polled Consumers”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-streams-overview-using-polled-consumers) for more details.

**Customizing Channel Names**

By using the @Input and @Output annotations, you can specify a customized channel name for the channel, as shown in the following example:

**public** **interface** Barista {

*@Input("inboundOrders")*

SubscribableChannel orders();

}

In the preceding example, the created bound channel is named inboundOrders.

Normally, you need not access individual channels or bindings directly (other then configuring them via @EnableBinding annotation). However there may be times, such as testing or other corner cases, when you do.

Aside from generating channels for each binding and registering them as Spring beans, for each bound interface, Spring Cloud Stream generates a bean that implements the interface. That means you can have access to the interfaces representing the bindings or individual channels by auto-wiring either in your application, as shown in the following two examples:

*Autowire Binding interface*

*@Autowire*

**private** Source source

**public** **void** sayHello(String name) {

source.output().send(MessageBuilder.withPayload(name).build());

}

*Autowire individual channel*

*@Autowire*

**private** MessageChannel output;

**public** **void** sayHello(String name) {

output.send(MessageBuilder.withPayload(name).build());

}

You can also use standard Spring’s @Qualifier annotation for cases when channel names are customized or in multiple-channel scenarios that require specifically named channels.

The following example shows how to use the @Qualifier annotation in this way:

*@Autowire*

*@Qualifier("myChannel")*

**private** MessageChannel output;

**27.3 Producing and Consuming Messages**

You can write a Spring Cloud Stream application by using either Spring Integration annotations or Spring Cloud Stream native annotation.

**27.3.1 Spring Integration Support**

Spring Cloud Stream is built on the concepts and patterns defined by [Enterprise Integration Patterns](http://www.enterpriseintegrationpatterns.com/) and relies in its internal implementation on an already established and popular implementation of Enterprise Integration Patterns within the Spring portfolio of projects: [Spring Integration](https://projects.spring.io/spring-integration/) framework.

So its only natiural for it to support the foundation, semantics, and configuration options that are already established by Spring Integration

For example, you can attach the output channel of a Source to a MessageSource and use the familiar @InboundChannelAdapter annotation, as follows:

*@EnableBinding(Source.class)*

**public** **class** TimerSource {

*@Bean*

*@InboundChannelAdapter(value = Source.OUTPUT, poller = @Poller(fixedDelay = "10", maxMessagesPerPoll = "1"))*

**public** MessageSource<String> timerMessageSource() {

**return** () -> **new** GenericMessage<>("Hello Spring Cloud Stream");

}

}

Similarly, you can use @Transformer or @ServiceActivator while providing an implementation of a message handler method for a *Processor* binding contract, as shown in the following example:

*@EnableBinding(Processor.class)*

**public** **class** TransformProcessor {

*@Transformer(inputChannel = Processor.INPUT, outputChannel = Processor.OUTPUT)*

**public** Object transform(String message) {

**return** message.toUpperCase();

}

}

|  |
| --- |
| [Note] |
| While this may be skipping ahead a bit, it is important to understand that, when you consume from the same binding using @StreamListener annotation, a pub-sub model is used. Each method annotated with @StreamListener receives its own copy of a message, and each one has its own consumer group. However, if you consume from the same binding by using one of the Spring Integration annotation (such as @Aggregator, @Transformer, or @ServiceActivator), those consume in a competing model. No individual consumer group is created for each subscription. |

**27.3.2 Using @StreamListener Annotation**

Complementary to its Spring Integration support, Spring Cloud Stream provides its own @StreamListener annotation, modeled after other Spring Messaging annotations (@MessageMapping, @JmsListener, @RabbitListener, and others) and provides conviniences, such as content-based routing and others.

*@EnableBinding(Sink.class)*

**public** **class** VoteHandler {

*@Autowired*

VotingService votingService;

*@StreamListener(Sink.INPUT)*

**public** **void** handle(Vote vote) {

votingService.record(vote);

}

}

As with other Spring Messaging methods, method arguments can be annotated with @Payload, @Headers, and @Header.

For methods that return data, you must use the @SendTo annotation to specify the output binding destination for data returned by the method, as shown in the following example:

*@EnableBinding(Processor.class)*

**public** **class** TransformProcessor {

*@Autowired*

VotingService votingService;

*@StreamListener(Processor.INPUT)*

*@SendTo(Processor.OUTPUT)*

**public** VoteResult handle(Vote vote) {

**return** votingService.record(vote);

}

}

**27.3.3 Using @StreamListener for Content-based routing**

Spring Cloud Stream supports dispatching messages to multiple handler methods annotated with @StreamListener based on conditions.

In order to be eligible to support conditional dispatching, a method must satisfy the follow conditions:

* It must not return a value.
* It must be an individual message handling method (reactive API methods are not supported).

The condition is specified by a SpEL expression in the condition argument of the annotation and is evaluated for each message. All the handlers that match the condition are invoked in the same thread, and no assumption must be made about the order in which the invocations take place.

In the following example of a @StreamListener with dispatching conditions, all the messages bearing a header type with the value bogey are dispatched to thereceiveBogey method, and all the messages bearing a header type with the value bacall are dispatched to the receiveBacall method.

*@EnableBinding(Sink.class)*

*@EnableAutoConfiguration*

**public** **static** **class** TestPojoWithAnnotatedArguments {

*@StreamListener(target = Sink.INPUT, condition = "headers['type']=='bogey'")*

**public** **void** receiveBogey(*@Payload* BogeyPojo bogeyPojo) {

*// handle the message*

}

*@StreamListener(target = Sink.INPUT, condition = "headers['type']=='bacall'")*

**public** **void** receiveBacall(*@Payload* BacallPojo bacallPojo) {

*// handle the message*

}

}

**Content Type Negotiation in the Context of condition**

It is important to understand some of the mechanics behind content-based routing using the condition argument of @StreamListener, especially in the context of the type of the message as a whole. It may also help if you familiarize yourself with the [Chapter 30, *Content Type Negotiation*](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#content-type-management) before you proceed.

Consider the following scenario:

*@EnableBinding(Sink.class)*

*@EnableAutoConfiguration*

**public** **static** **class** CatsAndDogs {

*@StreamListener(target = Sink.INPUT, condition = "payload.class.simpleName=='Dog'")*

**public** **void** bark(Dog dog) {

*// handle the message*

}

*@StreamListener(target = Sink.INPUT, condition = "payload.class.simpleName=='Cat'")*

**public** **void** purr(Cat cat) {

*// handle the message*

}

}

The preceding code is perfectly valid. It compiles and deploys without any issues, yet it never produces the result you expect.

That is because you are testing something that does not yet exist in a state you expect. That is becouse the payload of the message is not yet converted from the wire format (byte[]) to the desired type. In other words, it has not yet gone through the type conversion process described in the [Chapter 30, *Content Type Negotiation*](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#content-type-management).

So, unless you use a SPeL expression that evaluates raw data (for example, the value of the first byte in the byte array), use message header-based expressions (such as condition = "headers['type']=='dog'").

|  |
| --- |
| [Note] |
| At the moment, dispatching through @StreamListener conditions is supported only for channel-based binders (not for reactive programming) support. |

**27.3.4 Using Polled Consumers**

When using polled consumers, you poll the PollableMessageSource on demand. Consider the following example of a polled consumer:

**public** **interface** PolledConsumer {

*@Input*

PollableMessageSource destIn();

*@Output*

MessageChannel destOut();

}

Given the polled consumer in the preceding example, you might use it as follows:

*@Bean*

**public** ApplicationRunner poller(PollableMessageSource destIn, MessageChannel destOut) {

**return** args -> {

**while** (someCondition()) {

**try** {

**if** (!destIn.poll(m -> {

String newPayload = ((String) m.getPayload()).toUpperCase();

destOut.send(**new** GenericMessage<>(newPayload));

})) {

Thread.sleep(1000);

}

}

**catch** (Exception e) {

*// handle failure (throw an exception to reject the message);*

}

}

};

}

The PollableMessageSource.poll() method takes a MessageHandler argument (often a lambda expression, as shown here). It returns true if the message was received and successfully processed.

As with message-driven consumers, if the MessageHandler throws an exception, messages are published to error channels, as discussed in “[???](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html)”.

Normally, the poll() method acknowledges the message when the MessageHandler exits. If the method exits abnormally, the message is rejected (not re-queued). You can override that behavior by taking responsibility for the acknowledgment, as shown in the following example:

*@Bean*

**public** ApplicationRunner poller(PollableMessageSource dest1In, MessageChannel dest2Out) {

**return** args -> {

**while** (someCondition()) {

**if** (!dest1In.poll(m -> {

StaticMessageHeaderAccessor.getAcknowledgmentCallback(m).noAutoAck();

*// e.g. hand off to another thread which can perform the ack*

*// or acknowledge(Status.REQUEUE)*

})) {

Thread.sleep(1000);

}

}

};

}

|  |  |  |
| --- | --- | --- |
| [Important] | | **Important** |
| You must ack (or nack) the message at some point, to avoid resource leaks. |
| [Important] | **Important** | | |
| Some messaging systems (such as Apache Kafka) maintain a simple offset in a log. If a delivery fails and is re-queued with StaticMessageHeaderAccessor.getAcknowledgmentCallback(m).acknowledge(Status.REQUEUE);, any later successfully ack’d messages are redelivered. | | |

There is also an overloaded poll method, for which the definition is as follows:

poll(MessageHandler handler, ParameterizedTypeReference<?> type)

The type is a conversion hint that allows the incoming message payload to be converted, as shown in the following example:

**boolean** result = pollableSource.poll(received -> {

Map<String, Foo> payload = (Map<String, Foo>) received.getPayload();

...

}, **new** ParameterizedTypeReference<Map<String, Foo>>() {});

**27.4 Error Handling**

Errors happen, and Spring Cloud Stream provides several flexible mechanisms to handle them. The error handling comes in two flavors:

* **application:** The error handling is done within the application (custom error handler).
* **system:** The error handling is delegated to the binder (re-queue, DL, and others). Note that the techniques are dependent on binder implementation and the capability of the underlying messaging middleware.

Spring Cloud Stream uses the [Spring Retry](https://github.com/spring-projects/spring-retry) library to facilitate successful message processing. See [Section 27.4.3, “Retry Template”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_retry_template) for more details. However, when all fails, the exceptions thrown by the message handlers are propagated back to the binder. At that point, binder invokes custom error handler or communicates the error back to the messaging system (re-queue, DLQ, and others).

**27.4.1 Application Error Handling**

There are two types of application-level error handling. Errors can be handled at each binding subscription or a global handler can handle all the binding subscription errors. Let’s review the details.

**Figure 27.1. A Spring Cloud Stream Sink Application with Custom and Global Error Handlers**

For each input binding, Spring Cloud Stream creates a dedicated error channel with the following semantics <destinationName>.errors.

|  |
| --- |
| [Note] |
| The <destinationName> consists of the name of the binding (such as input) and the name of the group (such as myGroup). |

Consider the following:

spring.cloud.stream.bindings.input.group=myGroup

*@StreamListener(Sink.INPUT)* *// destination name 'input.myGroup'*

**public** **void** handle(Person value) {

**throw** **new** RuntimeException("BOOM!");

}

*@ServiceActivator(inputChannel = Processor.INPUT + ".myGroup.errors")* *//channel name 'input.myGroup.errors'*

**public** **void** error(Message<?> message) {

System.out.println("Handling ERROR: " + message);

}

In the preceding example the destination name is input.myGroup and the dedicated error channel name is input.myGroup.errors.

|  |
| --- |
| [Note] |
| The use of @StreamListener annotation is intended specifically to define bindings that bridge internal channels and external destinations. Given that the destination specific error channel does NOT have an associated external destination, such channel is a prerogative of Spring Integration (SI). This means that the handler for such destination must be defined using one of the SI handler annotations (i.e., @ServiceActivator, @Transformer etc.). |
| [Note] |
| If group is not specified anonymous group is used (something like input.anonymous.2K37rb06Q6m2r51-SPIDDQ), which is not suitable for error handling scenarious, since you don’t know what it’s going to be until the destination is created. | |

Also, in the event you are binding to the existing destination such as:

spring.cloud.stream.bindings.input.destination=myFooDestination

spring.cloud.stream.bindings.input.group=myGroup

the full destination name is myFooDestination.myGroup and then the dedicated error channel name is myFooDestination.myGroup.errors.

Back to the example…​

The handle(..) method, which subscribes to the channel named input, throws an exception. Given there is also a subscriber to the error channel input.myGroup.errors all error messages are handled by this subscriber.

If you have multiple bindings, you may want to have a single error handler. Spring Cloud Stream automatically provides support for a *global error channel* by bridging each individual error channel to the channel named errorChannel, allowing a single subscriber to handle all errors, as shown in the following example:

*@StreamListener("errorChannel")*

**public** **void** error(Message<?> message) {

System.out.println("Handling ERROR: " + message);

}

This may be a convenient option if error handling logic is the same regardless of which handler produced the error.

Also, error messages sent to the errorChannel can be published to the specific destination at the broker by configuring a binding named error for the outbound target. This option provides a mechanism to automatically send error messages to another application bound to that destination or for later retrieval (for example, audit). For example, to publish error messages to a broker destination named myErrors, set the following property:

spring.cloud.stream.bindings.error.destination=myErrors.

|  |
| --- |
| [Note] |
| The ability to bridge global error channel to a broker destination essentially provides a mechanism which connects the *application-level* error handling with the *system-level* error handling. |

**27.4.2 System Error Handling**

System-level error handling implies that the errors are communicated back to the messaging system and, given that not every messaging system is the same, the capabilities may differ from binder to binder.

That said, in this section we explain the general idea behind system level error handling and use Rabbit binder as an example. NOTE: Kafka binder provides similar support, although some configuration properties do differ. Also, for more details and configuration options, see the individual binder’s documentation.

If no internal error handlers are configured, the errors propagate to the binders, and the binders subsequently propagate those errors back to the messaging system. Depending on the capabilities of the messaging system such a system may *drop* the message, *re-queue* the message for re-processing or *send the failed message to DLQ*. Both Rabbit and Kafka support these concepts. However, other binders may not, so refer to your individual binder’s documentation for details on supported system-level error-handling options.

**Drop Failed Messages**

By default, if no additional system-level configuration is provided, the messaging system drops the failed message. While acceptable in some cases, for most cases, it is not, and we need some recovery mechanism to avoid message loss.

**DLQ - Dead Letter Queue**

DLQ allows failed messages to be sent to a special destination: - *Dead Letter Queue*.

When configured, failed messages are sent to this destination for subsequent re-processing or auditing and reconciliation.

For example, continuing on the previous example and to set up the DLQ with Rabbit binder, you need to set the following property:

spring.cloud.stream.rabbit.bindings.input.consumer.auto-bind-dlq=true

Keep in mind that, in the above property, input corresponds to the name of the input destination binding. The consumer indicates that it is a consumer property and auto-bind-dlq instructs the binder to configure DLQ for input destination, which results in an additional Rabbit queue named input.myGroup.dlq.

Once configured, all failed messages are routed to this queue with an error message similar to the following:

delivery\_mode: 1

headers:

x-death:

count: 1

reason: rejected

queue: input.hello

time: 1522328151

exchange:

routing-keys: input.myGroup

Payload {"name”:"Bob"}

As you can see from the above, your original message is preserved for further actions.

However, one thing you may have noticed is that there is limited information on the original issue with the message processing. For example, you do not see a stack trace corresponding to the original error. To get more relevant information about the original error, you must set an additional property:

spring.cloud.stream.rabbit.bindings.input.consumer.republish-to-dlq=true

Doing so forces the internal error handler to intercept the error message and add additional information to it before publishing it to DLQ. Once configured, you can see that the error message contains more information relevant to the original error, as follows:

delivery\_mode: 2

headers:

x-original-exchange:

x-exception-message: has an error

x-original-routingKey: input.myGroup

x-exception-stacktrace: org.springframework.messaging.MessageHandlingException: nested exception is

org.springframework.messaging.MessagingException: has an error, failedMessage=GenericMessage [payload=byte[15],

headers={amqp\_receivedDeliveryMode=NON\_PERSISTENT, amqp\_receivedRoutingKey=input.hello, amqp\_deliveryTag=1,

deliveryAttempt=3, amqp\_consumerQueue=input.hello, amqp\_redelivered=false, id=a15231e6-3f80-677b-5ad7-d4b1e61e486e,

amqp\_consumerTag=amq.ctag-skBFapilvtZhDsn0k3ZmQg, contentType=application/json, timestamp=1522327846136}]

at org.spring...integ...han...MethodInvokingMessageProcessor.processMessage(MethodInvokingMessageProcessor.java:107)

at. . . . .

Payload {"name”:"Bob"}

This effectively combines application-level and system-level error handling to further assist with downstream troubleshooting mechanics.

**Re-queue Failed Messages**

As mentioned earlier, the currently supported binders (Rabbit and Kafka) rely on RetryTemplate to facilitate successful message processing. See [Section 27.4.3, “Retry Template”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_retry_template) for details. However, for cases when max-attempts property is set to 1, internal reprocessing of the message is disabled. At this point, you can facilitate message re-processing (re-tries) by instructing the messaging system to re-queue the failed message. Once re-queued, the failed message is sent back to the original handler, essentially creating a retry loop.

This option may be feasible for cases where the nature of the error is related to some sporadic yet short-term unavailability of some resource.

To accomplish that, you must set the following properties:

spring.cloud.stream.bindings.input.consumer.max-attempts=1

spring.cloud.stream.rabbit.bindings.input.consumer.requeue-rejected=true

In the preceding example, the max-attempts set to 1 essentially disabling internal re-tries and requeue-rejected (short for *requeue rejected messages*) is set to true. Once set, the failed message is resubmitted to the same handler and loops continuously or until the handler throws AmqpRejectAndDontRequeueExceptionessentially allowing you to build your own re-try logic within the handler itself.

**27.4.3 Retry Template**

The RetryTemplate is part of the [Spring Retry](https://github.com/spring-projects/spring-retry) library. While it is out of scope of this document to cover all of the capabilities of the RetryTemplate, we will mention the following consumer properties that are specifically related to the RetryTemplate:

maxAttempts

The number of attempts to process the message.

Default: 3.

backOffInitialInterval

The backoff initial interval on retry.

Default 1000 milliseconds.

backOffMaxInterval

The maximum backoff interval.

Default 10000 milliseconds.

backOffMultiplier

The backoff multiplier.

Default 2.0.

While the preceding settings are sufficient for majority of the customization requirements, they may not satisfy certain complex requirements at, which point you may want to provide your own instance of the RetryTemplate. To do so configure it as a bean in your application configuration. The application provided instance will override the one provided by the framework. Also, to avoid conflicts you must qualify the instance of the RetryTemplate you want to be used by the binder as @StreamRetryTemplate. For example,

*@StreamRetryTemplate*

**public** RetryTemplate myRetryTemplate() {

**return** **new** RetryTemplate();

}

As you can see from the above example you don’t need to annotate it with @Bean since @StreamRetryTemplate is a qualified @Bean.

**27.5 Reactive Programming Support**

Spring Cloud Stream also supports the use of reactive APIs where incoming and outgoing data is handled as continuous data flows. Support for reactive APIs is available through spring-cloud-stream-reactive, which needs to be added explicitly to your project.

The programming model with reactive APIs is declarative. Instead of specifying how each individual message should be handled, you can use operators that describe functional transformations from inbound to outbound data flows.

At present Spring Cloud Stream supports the only the [Reactor API](https://projectreactor.io/). In the future, we intend to support a more generic model based on Reactive Streams.

The reactive programming model also uses the @StreamListener annotation for setting up reactive handlers. The differences are that:

* The @StreamListener annotation must not specify an input or output, as they are provided as arguments and return values from the method.
* The arguments of the method must be annotated with @Input and @Output, indicating which input or output the incoming and outgoing data flows connect to, respectively.
* The return value of the method, if any, is annotated with @Output, indicating the input where data should be sent.

|  |
| --- |
| [Note] |
| Reactive programming support requires Java 1.8. |
| [Note] |
| As of Spring Cloud Stream 1.1.1 and later (starting with release train Brooklyn.SR2), reactive programming support requires the use of Reactor 3.0.4.RELEASE and higher. Earlier Reactor versions (including 3.0.1.RELEASE, 3.0.2.RELEASE and 3.0.3.RELEASE) are not supported.spring-cloud-stream-reactive transitively retrieves the proper version, but it is possible for the project structure to manage the version of the io.projectreactor:reactor-core to an earlier release, especially when using Maven. This is the case for projects generated by using Spring Initializr with Spring Boot 1.x, which overrides the Reactor version to 2.0.8.RELEASE. In such cases, you must ensure that the proper version of the artifact is released. You can do so by adding a direct dependency on io.projectreactor:reactor-core with a version of 3.0.4.RELEASE or later to your project. | |

|  |
| --- |
| [Note] |
| The use of term, “reactive”, currently refers to the reactive APIs being used and not to the execution model being reactive (that is, the bound endpoints still use a 'push' rather than a 'pull' model). While some backpressure support is provided by the use of Reactor, we do intend, in a future release, to support entirely reactive pipelines by the use of native reactive clients for the connected middleware. |

**27.5.1 Reactor-based Handlers**

A Reactor-based handler can have the following argument types:

* For arguments annotated with @Input, it supports the Reactor Flux type. The parameterization of the inbound Flux follows the same rules as in the case of individual message handling: It can be the entire Message, a POJO that can be the Message payload, or a POJO that is the result of a transformation based on the Message content-type header. Multiple inputs are provided.
* For arguments annotated with Output, it supports the FluxSender type, which connects a Flux produced by the method with an output. Generally speaking, specifying outputs as arguments is only recommended when the method can have multiple outputs.

A Reactor-based handler supports a return type of Flux. In that case, it must be annotated with @Output. We recommend using the return value of the method when a single output Flux is available.

The following example shows a Reactor-based Processor:

*@EnableBinding(Processor.class)*

*@EnableAutoConfiguration*

**public** **static** **class** UppercaseTransformer {

*@StreamListener*

*@Output(Processor.OUTPUT)*

**public** Flux<String> receive(*@Input(Processor.INPUT)* Flux<String> input) {

**return** input.map(s -> s.toUpperCase());

}

}

The same processor using output arguments looks like the following example:

*@EnableBinding(Processor.class)*

*@EnableAutoConfiguration*

**public** **static** **class** UppercaseTransformer {

*@StreamListener*

**public** **void** receive(*@Input(Processor.INPUT)* Flux<String> input,

*@Output(Processor.OUTPUT)* FluxSender output) {

output.send(input.map(s -> s.toUpperCase()));

}

}

**27.5.2 Reactive Sources**

Spring Cloud Stream reactive support also provides the ability for creating reactive sources through the @StreamEmitter annotation. By using the @StreamEmitterannotation, a regular source may be converted to a reactive one. @StreamEmitter is a method level annotation that marks a method to be an emitter to outputs declared with @EnableBinding. You cannot use the @Input annotation along with @StreamEmitter, as the methods marked with this annotation are not listening for any input. Rather, methods marked with @StreamEmitter generate output. Following the same programming model used in @StreamListener, @StreamEmitteralso allows flexible ways of using the @Output annotation, depending on whether the method has any arguments, a return type, and other considerations.

The remainder of this section contains examples of using the @StreamEmitter annotation in various styles.

The following example emits the Hello, World message every millisecond and publishes to a Reactor Flux:

*@EnableBinding(Source.class)*

*@EnableAutoConfiguration*

**public** **static** **class** HelloWorldEmitter {

*@StreamEmitter*

*@Output(Source.OUTPUT)*

**public** Flux<String> emit() {

**return** Flux.intervalMillis(1)

.map(l -> "Hello World");

}

}

In the preceding example, the resulting messages in the Flux are sent to the output channel of the Source.

The next example is another flavor of an @StreamEmmitter that sends a Reactor Flux. Instead of returning a Flux, the following method uses a FluxSender to programmatically send a Flux from a source:

*@EnableBinding(Source.class)*

*@EnableAutoConfiguration*

**public** **static** **class** HelloWorldEmitter {

*@StreamEmitter*

*@Output(Source.OUTPUT)*

**public** **void** emit(FluxSender output) {

output.send(Flux.intervalMillis(1)

.map(l -> "Hello World"));

}

}

The next example is exactly same as the above snippet in functionality and style. However, instead of using an explicit @Output annotation on the method, it uses the annotation on the method parameter.

*@EnableBinding(Source.class)*

*@EnableAutoConfiguration*

**public** **static** **class** HelloWorldEmitter {

*@StreamEmitter*

**public** **void** emit(*@Output(Source.OUTPUT)* FluxSender output) {

output.send(Flux.intervalMillis(1)

.map(l -> "Hello World"));

}

}

The last example in this section is yet another flavor of writing reacting sources by using the Reactive Streams Publisher API and taking advantage of the support for it in [Spring Integration Java DSL](https://github.com/spring-projects/spring-integration-java-dsl/wiki/Spring-Integration-Java-DSL-Reference). The Publisher in the following example still uses Reactor Flux under the hood, but, from an application perspective, that is transparent to the user and only needs Reactive Streams and Java DSL for Spring Integration:

*@EnableBinding(Source.class)*

*@EnableAutoConfiguration*

**public** **static** **class** HelloWorldEmitter {

*@StreamEmitter*

*@Output(Source.OUTPUT)*

*@Bean*

**public** Publisher<Message<String>> emit() {

**return** IntegrationFlows.from(() ->

**new** GenericMessage<>("Hello World"),

e -> e.poller(p -> p.fixedDelay(1)))

.toReactivePublisher();

}

}

**28. Binders**

Spring Cloud Stream provides a Binder abstraction for use in connecting to physical destinations at the external middleware. This section provides information about the main concepts behind the Binder SPI, its main components, and implementation-specific details.

**28.1 Producers and Consumers**

The following image shows the general relationship of producers and consumers:

**Figure 28.1. Producers and Consumers**

A producer is any component that sends messages to a channel. The channel can be bound to an external message broker with a Binder implementation for that broker. When invoking the bindProducer() method, the first parameter is the name of the destination within the broker, the second parameter is the local channel instance to which the producer sends messages, and the third parameter contains properties (such as a partition key expression) to be used within the adapter that is created for that channel.

A consumer is any component that receives messages from a channel. As with a producer, the consumer’s channel can be bound to an external message broker. When invoking the bindConsumer() method, the first parameter is the destination name, and a second parameter provides the name of a logical group of consumers. Each group that is represented by consumer bindings for a given destination receives a copy of each message that a producer sends to that destination (that is, it follows normal publish-subscribe semantics). If there are multiple consumer instances bound with the same group name, then messages are load-balanced across those consumer instances so that each message sent by a producer is consumed by only a single consumer instance within each group (that is, it follows normal queueing semantics).

**28.2 Binder SPI**

The Binder SPI consists of a number of interfaces, out-of-the box utility classes, and discovery strategies that provide a pluggable mechanism for connecting to external middleware.

The key point of the SPI is the Binder interface, which is a strategy for connecting inputs and outputs to external middleware. The following listing shows the definnition of the Binder interface:

**public** **interface** Binder<T, C **extends** ConsumerProperties, P **extends** ProducerProperties> {

Binding<T> bindConsumer(String name, String group, T inboundBindTarget, C consumerProperties);

Binding<T> bindProducer(String name, T outboundBindTarget, P producerProperties);

}

The interface is parameterized, offering a number of extension points:

* Input and output bind targets. As of version 1.0, only MessageChannel is supported, but this is intended to be used as an extension point in the future.
* Extended consumer and producer properties, allowing specific Binder implementations to add supplemental properties that can be supported in a type-safe manner.

A typical binder implementation consists of the following:

* A class that implements the Binder interface;
* A Spring @Configuration class that creates a bean of type Binder along with the middleware connection infrastructure.
* A META-INF/spring.binders file found on the classpath containing one or more binder definitions, as shown in the following example:
* kafka:\

org.springframework.cloud.stream.binder.kafka.config.KafkaBinderConfiguration

**28.3 Binder Detection**

Spring Cloud Stream relies on implementations of the Binder SPI to perform the task of connecting channels to message brokers. Each Binder implementation typically connects to one type of messaging system.

**28.3.1 Classpath Detection**

By default, Spring Cloud Stream relies on Spring Boot’s auto-configuration to configure the binding process. If a single Binder implementation is found on the classpath, Spring Cloud Stream automatically uses it. For example, a Spring Cloud Stream project that aims to bind only to RabbitMQ can add the following dependency:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-stream-binder-rabbit</artifactId>

</dependency>

For the specific Maven coordinates of other binder dependencies, see the documentation of that binder implementation.

**28.4 Multiple Binders on the Classpath**

When multiple binders are present on the classpath, the application must indicate which binder is to be used for each channel binding. Each binder configuration contains a META-INF/spring.binders file, which is a simple properties file, as shown in the following example:

rabbit:\

org.springframework.cloud.stream.binder.rabbit.config.RabbitServiceAutoConfiguration

Similar files exist for the other provided binder implementations (such as Kafka), and custom binder implementations are expected to provide them as well. The key represents an identifying name for the binder implementation, whereas the value is a comma-separated list of configuration classes that each contain one and only one bean definition of type org.springframework.cloud.stream.binder.Binder.

Binder selection can either be performed globally, using the spring.cloud.stream.defaultBinder property (for example, spring.cloud.stream.defaultBinder=rabbit) or individually, by configuring the binder on each channel binding. For instance, a processor application (that has channels named input and output for read and write respectively) that reads from Kafka and writes to RabbitMQ can specify the following configuration:

spring.cloud.stream.bindings.input.binder=kafka

spring.cloud.stream.bindings.output.binder=rabbit

**28.5 Connecting to Multiple Systems**

By default, binders share the application’s Spring Boot auto-configuration, so that one instance of each binder found on the classpath is created. If your application should connect to more than one broker of the same type, you can specify multiple binder configurations, each with different environment settings.

|  |
| --- |
| [Note] |
| Turning on explicit binder configuration disables the default binder configuration process altogether. If you do so, all binders in use must be included in the configuration. Frameworks that intend to use Spring Cloud Stream transparently may create binder configurations that can be referenced by name, but they do not affect the default binder configuration. In order to do so, a binder configuration may have its defaultCandidate flag set to false (for example, spring.cloud.stream.binders.<configurationName>.defaultCandidate=false). This denotes a configuration that exists independently of the default binder configuration process. |

The following example shows a typical configuration for a processor application that connects to two RabbitMQ broker instances:

spring:

cloud:

stream:

bindings:

input:

destination: thing1

binder: rabbit1

output:

destination: thing2

binder: rabbit2

binders:

rabbit1:

type: rabbit

environment:

spring:

rabbitmq:

host: <host1>

rabbit2:

type: rabbit

environment:

spring:

rabbitmq:

host: <host2>

**28.6 Binding visualization and control**

Since version 2.0, Spring Cloud Stream supports visualization and control of the Bindings through Actuator endpoints.

Starting with version 2.0 actuator and web are optional, you must first add one of the web dependencies as well as add the actuator dependency manually. The following example shows how to add the dependency for the Web framework:

<dependency>

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

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

</dependency>

The following example shows how to add the dependency for the WebFlux framework:

<dependency>

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

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

</dependency>

You can add the Actuator dependency as follows:

<dependency>

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

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

</dependency>

|  |
| --- |
| [Note] |
| To run Spring Cloud Stream 2.0 apps in Cloud Foundry, you must add spring-boot-starter-web and spring-boot-starter-actuator to the classpath. Otherwise, the application will not start due to health check failures. |

You must also enable the bindings actuator endpoints by setting the following property: --management.endpoints.web.exposure.include=bindings.

Once those prerequisites are satisfied. you should see the following in the logs when application start:

: Mapped "{[/actuator/bindings/{name}],methods=[POST]. . .

: Mapped "{[/actuator/bindings],methods=[GET]. . .

: Mapped "{[/actuator/bindings/{name}],methods=[GET]. . .

To visualize the current bindings, access the following URL: http://<host>:<port>/actuator/bindings

Alternative, to see a single binding, access one of the URLs similar to the following: http://<host>:<port>/actuator/bindings/myBindingName

You can also stop, start, pause, and resume individual bindings by posting to the same URL while providing a state argument as JSON, as shown in the following examples:

curl -d '{"state":"STOPPED"}' -H "Content-Type: application/json" -X POST http://<host>:<port>/actuator/bindings/myBindingName curl -d '{"state":"STARTED"}' -H "Content-Type: application/json" -X POST http://<host>:<port>/actuator/bindings/myBindingName curl -d '{"state":"PAUSED"}' -H "Content-Type: application/json" -X POST http://<host>:<port>/actuator/bindings/myBindingName curl -d '{"state":"RESUMED"}' -H "Content-Type: application/json" -X POST http://<host>:<port>/actuator/bindings/myBindingName

|  |
| --- |
| [Note] |
| PAUSED and RESUMED work only when the corresponding binder and its underlying technology supports it. Otherwise, you see the warning message in the logs. Currently, only Kafka binder supports the PAUSED and RESUMED states. |

**28.7 Binder Configuration Properties**

The following properties are available when customizing binder configurations. These properties exposed via org.springframework.cloud.stream.config.BinderProperties

They must be prefixed with spring.cloud.stream.binders.<configurationName>.

type

The binder type. It typically references one of the binders found on the classpath — in particular, a key in a META-INF/spring.binders file.

By default, it has the same value as the configuration name.

inheritEnvironment

Whether the configuration inherits the environment of the application itself.

Default: true.

environment

Root for a set of properties that can be used to customize the environment of the binder. When this property is set, the context in which the binder is being created is not a child of the application context. This setting allows for complete separation between the binder components and the application components.

Default: empty.

defaultCandidate

Whether the binder configuration is a candidate for being considered a default binder or can be used only when explicitly referenced. This setting allows adding binder configurations without interfering with the default processing.

Default: true.

**29. Configuration Options**

Spring Cloud Stream supports general configuration options as well as configuration for bindings and binders. Some binders let additional binding properties support middleware-specific features.

Configuration options can be provided to Spring Cloud Stream applications through any mechanism supported by Spring Boot. This includes application arguments, environment variables, and YAML or .properties files.

**29.1 Binding Service Properties**

These properties are exposed via org.springframework.cloud.stream.config.BindingServiceProperties

spring.cloud.stream.instanceCount

The number of deployed instances of an application. Must be set for partitioning on the producer side. Must be set on the consumer side when using RabbitMQ and with Kafka if autoRebalanceEnabled=false.

Default: 1.

spring.cloud.stream.instanceIndex

The instance index of the application: A number from 0 to instanceCount - 1. Used for partitioning with RabbitMQ and with Kafka if autoRebalanceEnabled=false. Automatically set in Cloud Foundry to match the application’s instance index.

spring.cloud.stream.dynamicDestinations

A list of destinations that can be bound dynamically (for example, in a dynamic routing scenario). If set, only listed destinations can be bound.

Default: empty (letting any destination be bound).

spring.cloud.stream.defaultBinder

The default binder to use, if multiple binders are configured. See [Multiple Binders on the Classpath](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#multiple-binders).

Default: empty.

spring.cloud.stream.overrideCloudConnectors

This property is only applicable when the cloud profile is active and Spring Cloud Connectors are provided with the application. If the property is false (the default), the binder detects a suitable bound service (for example, a RabbitMQ service bound in Cloud Foundry for the RabbitMQ binder) and uses it for creating connections (usually through Spring Cloud Connectors). When set to true, this property instructs binders to completely ignore the bound services and rely on Spring Boot properties (for example, relying on the spring.rabbitmq.\* properties provided in the environment for the RabbitMQ binder). The typical usage of this property is to be nested in a customized environment [when connecting to multiple systems](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#multiple-systems).

Default: false.

spring.cloud.stream.bindingRetryInterval

The interval (in seconds) between retrying binding creation when, for example, the binder does not support late binding and the broker (for example, Apache Kafka) is down. Set it to zero to treat such conditions as fatal, preventing the application from starting.

Default: 30

**29.2 Binding Properties**

Binding properties are supplied by using the format of spring.cloud.stream.bindings.<channelName>.<property>=<value>. The <channelName> represents the name of the channel being configured (for example, output for a Source).

To avoid repetition, Spring Cloud Stream supports setting values for all channels, in the format of spring.cloud.stream.default.<property>=<value>.

In what follows, we indicate where we have omitted the spring.cloud.stream.bindings.<channelName>. prefix and focus just on the property name, with the understanding that the prefix ise included at runtime.

**29.2.1 Common Binding Properties**

These properties are exposed via org.springframework.cloud.stream.config.BindingProperties

The following binding properties are available for both input and output bindings and must be prefixed with spring.cloud.stream.bindings.<channelName>. (for example, spring.cloud.stream.bindings.input.destination=ticktock).

Default values can be set by using the spring.cloud.stream.default prefix (for example`spring.cloud.stream.default.contentType=application/json`).

destination

The target destination of a channel on the bound middleware (for example, the RabbitMQ exchange or Kafka topic). If the channel is bound as a consumer, it could be bound to multiple destinations, and the destination names can be specified as comma-separated String values. If not set, the channel name is used instead. The default value of this property cannot be overridden.

group

The consumer group of the channel. Applies only to inbound bindings. See [Consumer Groups](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#consumer-groups).

Default: null (indicating an anonymous consumer).

contentType

The content type of the channel. See “[Chapter 30, *Content Type Negotiation*](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#content-type-management)”.

Default: null (no type coercion is performed).

binder

The binder used by this binding. See “[Section 28.4, “Multiple Binders on the Classpath”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#multiple-binders)” for details.

Default: null (the default binder is used, if it exists).

**29.2.2 Consumer Properties**

These properties are exposed via org.springframework.cloud.stream.binder.ConsumerProperties

The following binding properties are available for input bindings only and must be prefixed with spring.cloud.stream.bindings.<channelName>.consumer. (for example, spring.cloud.stream.bindings.input.consumer.concurrency=3).

Default values can be set by using the spring.cloud.stream.default.consumer prefix (for example, spring.cloud.stream.default.consumer.headerMode=none).

concurrency

The concurrency of the inbound consumer.

Default: 1.

partitioned

Whether the consumer receives data from a partitioned producer.

Default: false.

headerMode

When set to none, disables header parsing on input. Effective only for messaging middleware that does not support message headers natively and requires header embedding. This option is useful when consuming data from non-Spring Cloud Stream applications when native headers are not supported. When set to headers, it uses the middleware’s native header mechanism. When set to embeddedHeaders, it embeds headers into the message payload.

Default: depends on the binder implementation.

maxAttempts

If processing fails, the number of attempts to process the message (including the first). Set to 1 to disable retry.

Default: 3.

backOffInitialInterval

The backoff initial interval on retry.

Default: 1000.

backOffMaxInterval

The maximum backoff interval.

Default: 10000.

backOffMultiplier

The backoff multiplier.

Default: 2.0.

instanceIndex

When set to a value greater than equal to zero, it allows customizing the instance index of this consumer (if different from spring.cloud.stream.instanceIndex). When set to a negative value, it defaults to spring.cloud.stream.instanceIndex. See “[Section 32.2, “Instance Index and Instance Count”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-instance-index-instance-count)” for more information.

Default: -1.

instanceCount

When set to a value greater than equal to zero, it allows customizing the instance count of this consumer (if different from spring.cloud.stream.instanceCount). When set to a negative value, it defaults to spring.cloud.stream.instanceCount. See “[Section 32.2, “Instance Index and Instance Count”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-instance-index-instance-count)” for more information.

Default: -1.

useNativeDecoding

When set to true, the inbound message is deserialized directly by the client library, which must be configured correspondingly (for example, setting an appropriate Kafka producer value deserializer). When this configuration is being used, the inbound message unmarshalling is not based on the contentType of the binding. When native decoding is used, it is the responsibility of the producer to use an appropriate encoder (for example, the Kafka producer value serializer) to serialize the outbound message. Also, when native encoding and decoding is used, the headerMode=embeddedHeaders property is ignored and headers are not embedded in the message. See the producer property useNativeEncoding.

Default: false.

**29.2.3 Producer Properties**

These properties are exposed via org.springframework.cloud.stream.binder.ProducerProperties

The following binding properties are available for output bindings only and must be prefixed with spring.cloud.stream.bindings.<channelName>.producer. (for example, spring.cloud.stream.bindings.input.producer.partitionKeyExpression=payload.id).

Default values can be set by using the prefix spring.cloud.stream.default.producer (for example, spring.cloud.stream.default.producer.partitionKeyExpression=payload.id).

partitionKeyExpression

A SpEL expression that determines how to partition outbound data. If set, or if partitionKeyExtractorClass is set, outbound data on this channel is partitioned. partitionCount must be set to a value greater than 1 to be effective. Mutually exclusive with partitionKeyExtractorClass. See “[Section 26.6, “Partitioning Support”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#partitioning)”.

Default: null.

partitionKeyExtractorClass

A PartitionKeyExtractorStrategy implementation. If set, or if partitionKeyExpression is set, outbound data on this channel is partitioned. partitionCount must be set to a value greater than 1 to be effective. Mutually exclusive with partitionKeyExpression. See “[Section 26.6, “Partitioning Support”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#partitioning)”.

Default: null.

partitionSelectorClass

A PartitionSelectorStrategy implementation. Mutually exclusive with partitionSelectorExpression. If neither is set, the partition is selected as the hashCode(key) % partitionCount, where key is computed through either partitionKeyExpression or partitionKeyExtractorClass.

Default: null.

partitionSelectorExpression

A SpEL expression for customizing partition selection. Mutually exclusive with partitionSelectorClass. If neither is set, the partition is selected as the hashCode(key) % partitionCount, where key is computed through either partitionKeyExpression or partitionKeyExtractorClass.

Default: null.

partitionCount

The number of target partitions for the data, if partitioning is enabled. Must be set to a value greater than 1 if the producer is partitioned. On Kafka, it is interpreted as a hint. The larger of this and the partition count of the target topic is used instead.

Default: 1.

requiredGroups

A comma-separated list of groups to which the producer must ensure message delivery even if they start after it has been created (for example, by pre-creating durable queues in RabbitMQ).

headerMode

When set to none, it disables header embedding on output. It is effective only for messaging middleware that does not support message headers natively and requires header embedding. This option is useful when producing data for non-Spring Cloud Stream applications when native headers are not supported. When set to headers, it uses the middleware’s native header mechanism. When set to embeddedHeaders, it embeds headers into the message payload.

Default: Depends on the binder implementation.

useNativeEncoding

When set to true, the outbound message is serialized directly by the client library, which must be configured correspondingly (for example, setting an appropriate Kafka producer value serializer). When this configuration is being used, the outbound message marshalling is not based on the contentType of the binding. When native encoding is used, it is the responsibility of the consumer to use an appropriate decoder (for example, the Kafka consumer value de-serializer) to deserialize the inbound message. Also, when native encoding and decoding is used, the headerMode=embeddedHeaders property is ignored and headers are not embedded in the message. See the consumer property useNativeDecoding.

Default: false.

errorChannelEnabled

When set to true, if the binder supports asynchroous send results, send failures are sent to an error channel for the destination. See “[???](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html)” for more information.

Default: false.

**29.3 Using Dynamically Bound Destinations**

Besides the channels defined by using @EnableBinding, Spring Cloud Stream lets applications send messages to dynamically bound destinations. This is useful, for example, when the target destination needs to be determined at runtime. Applications can do so by using the BinderAwareChannelResolver bean, registered automatically by the @EnableBinding annotation.

The 'spring.cloud.stream.dynamicDestinations' property can be used for restricting the dynamic destination names to a known set (whitelisting). If this property is not set, any destination can be bound dynamically.

The BinderAwareChannelResolver can be used directly, as shown in the following example of a REST controller using a path variable to decide the target channel:

*@EnableBinding*

*@Controller*

**public** **class** SourceWithDynamicDestination {

*@Autowired*

**private** BinderAwareChannelResolver resolver;

*@RequestMapping(path = "/{target}", method = POST, consumes = "\*/\*")*

*@ResponseStatus(HttpStatus.ACCEPTED)*

**public** **void** handleRequest(*@RequestBody* String body, *@PathVariable("target")* target,

*@RequestHeader(HttpHeaders.CONTENT\_TYPE)* Object contentType) {

sendMessage(body, target, contentType);

}

**private** **void** sendMessage(String body, String target, Object contentType) {

resolver.resolveDestination(target).send(MessageBuilder.createMessage(body,

**new** MessageHeaders(Collections.singletonMap(MessageHeaders.CONTENT\_TYPE, contentType))));

}

}

Now consider what happens when we start the application on the default port (8080) and make the following requests with CURL:

curl -H "Content-Type: application/json" -X POST -d "customer-1" http://localhost:8080/customers

curl -H "Content-Type: application/json" -X POST -d "order-1" http://localhost:8080/orders

The destinations, 'customers' and 'orders', are created in the broker (in the exchange for Rabbit or in the topic for Kafka) with names of 'customers' and 'orders', and the data is published to the appropriate destinations.

The BinderAwareChannelResolver is a general-purpose Spring Integration DestinationResolver and can be injected in other components — for example, in a router using a SpEL expression based on the target field of an incoming JSON message. The following example includes a router that reads SpEL expressions:

*@EnableBinding*

*@Controller*

**public** **class** SourceWithDynamicDestination {

*@Autowired*

**private** BinderAwareChannelResolver resolver;

*@RequestMapping(path = "/", method = POST, consumes = "application/json")*

*@ResponseStatus(HttpStatus.ACCEPTED)*

**public** **void** handleRequest(*@RequestBody* String body, *@RequestHeader(HttpHeaders.CONTENT\_TYPE)* Object contentType) {

sendMessage(body, contentType);

}

**private** **void** sendMessage(Object body, Object contentType) {

routerChannel().send(MessageBuilder.createMessage(body,

**new** MessageHeaders(Collections.singletonMap(MessageHeaders.CONTENT\_TYPE, contentType))));

}

*@Bean(name = "routerChannel")*

**public** MessageChannel routerChannel() {

**return** **new** DirectChannel();

}

*@Bean*

*@ServiceActivator(inputChannel = "routerChannel")*

**public** ExpressionEvaluatingRouter router() {

ExpressionEvaluatingRouter router =

**new** ExpressionEvaluatingRouter(**new** SpelExpressionParser().parseExpression("payload.target"));

router.setDefaultOutputChannelName("default-output");

router.setChannelResolver(resolver);

**return** router;

}

}

The [Router Sink Application](https://github.com/spring-cloud-stream-app-starters/router) uses this technique to create the destinations on-demand.

If the channel names are known in advance, you can configure the producer properties as with any other destination. Alternatively, if you register a NewBindingCallback<> bean, it is invoked just before the binding is created. The callback takes the generic type of the extended producer properties used by the binder. It has one method:

**void** configure(String channelName, MessageChannel channel, ProducerProperties producerProperties,

T extendedProducerProperties);

The following example shows how to use the RabbitMQ binder:

*@Bean*

**public** NewBindingCallback<RabbitProducerProperties> dynamicConfigurer() {

**return** (name, channel, props, extended) -> {

props.setRequiredGroups("bindThisQueue");

extended.setQueueNameGroupOnly(true);

extended.setAutoBindDlq(true);

extended.setDeadLetterQueueName("myDLQ");

};

}

|  |
| --- |
| [Note] |
| If you need to support dynamic destinations with multiple binder types, use Object for the generic type and cast the extended argument as needed. |

**30. Content Type Negotiation**

Data transformation is one of the core features of any message-driven microservice architecture. Given that, in Spring Cloud Stream, such data is represented as a Spring Message, a message may have to be transformed to a desired shape or size before reaching its destination. This is required for two reasons:

1. To convert the contents of the incoming message to match the signature of the application-provided handler.
2. To convert the contents of the outgoing message to the wire format.

The wire format is typically byte[] (that is true for the Kafka and Rabbit binders), but it is governed by the binder implementation.

In Spring Cloud Stream, message transformation is accomplished with an org.springframework.messaging.converter.MessageConverter.

|  |
| --- |
| [Note] |
| As a supplement to the details to follow, you may also want to read the following [blog post](https://spring.io/blog/2018/02/26/spring-cloud-stream-2-0-content-type-negotiation-and-transformation). |

**30.1 Mechanics**

To better understand the mechanics and the necessity behind content-type negotiation, we take a look at a very simple use case by using the following message handler as an example:

*@StreamListener(Processor.INPUT)*

*@SendTo(Processor.OUTPUT)*

**public** String handle(Person person) {..}

|  |
| --- |
| [Note] |
| For simplicity, we assume that this is the only handler in the application (we assume there is no internal pipeline). |

The handler shown in the preceding example expects a Person object as an argument and produces a String type as an output. In order for the framework to succeed in passing the incoming Message as an argument to this handler, it has to somehow transform the payload of the Message type from the wire format to a Person type. In other words, the framework must locate and apply the appropriate MessageConverter. To accomplish that, the framework needs some instructions from the user. One of these instructions is already provided by the signature of the handler method itself (Person type). Consequently, in theory, that should be (and, in some cases, is) enough. However, for the majority of use cases, in order to select the appropriate MessageConverter, the framework needs an additional piece of information. That missing piece is contentType.

Spring Cloud Stream provides three mechanisms to define contentType (in order of precedence):

1. **HEADER**: The contentType can be communicated through the Message itself. By providing a contentType header, you declare the content type to use to locate and apply the appropriate MessageConverter.
2. **BINDING**: The contentType can be set per destination binding by setting the spring.cloud.stream.bindings.input.content-type property.

|  |
| --- |
| [Note] |
| The input segment in the property name corresponds to the actual name of the destination (which is “input” in our case). This approach lets you declare, on a per-binding basis, the content type to use to locate and apply the appropriate MessageConverter. |

1. **DEFAULT**: If contentType is not present in the Message header or the binding, the default application/json content type is used to locate and apply the appropriate MessageConverter.

As mentioned earlier, the preceding list also demonstrates the order of precedence in case of a tie. For example, a header-provided content type takes precedence over any other content type. The same applies for a content type set on a per-binding basis, which essentially lets you override the default content type. However, it also provides a sensible default (which was determined from community feedback).

Another reason for making application/json the default stems from the interoperability requirements driven by distributed microservices architectures, where producer and consumer not only run in different JVMs but can also run on different non-JVM platforms.

When the non-void handler method returns, if the the return value is already a Message, that Message becomes the payload. However, when the return value is not a Message, the new Message is constructed with the return value as the payload while inheriting headers from the input Message minus the headers defined or filtered by SpringIntegrationProperties.messageHandlerNotPropagatedHeaders. By default, there is only one header set there: contentType. This means that the new Message does not have contentType header set, thus ensuring that the contentType can evolve. You can always opt out of returning a Message from the handler method where you can inject any header you wish.

If there is an internal pipeline, the Message is sent to the next handler by going through the same process of conversion. However, if there is no internal pipeline or you have reached the end of it, the Message is sent back to the output destination.

**30.1.1 Content Type versus Argument Type**

As mentioned earlier, for the framework to select the appropriate MessageConverter, it requires argument type and, optionally, content type information. The logic for selecting the appropriate MessageConverter resides with the argument resolvers (HandlerMethodArgumentResolvers), which trigger right before the invocation of the user-defined handler method (which is when the actual argument type is known to the framework). If the argument type does not match the type of the current payload, the framework delegates to the stack of the pre-configured MessageConverters to see if any one of them can convert the payload. As you can see, the Object fromMessage(Message<?> message, Class<?> targetClass); operation of the MessageConverter takes targetClass as one of its arguments. The framework also ensures that the provided Message always contains a contentType header. When no contentType header was already present, it injects either the per-binding contentType header or the default contentType header. The combination of contentType argument type is the mechanism by which framework determines if message can be converted to a target type. If no appropriate MessageConverter is found, an exception is thrown, which you can handle by adding a custom MessageConverter (see “[Section 30.3, “User-defined Message Converters”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-user-defined-message-converters)”).

But what if the payload type matches the target type declared by the handler method? In this case, there is nothing to convert, and the payload is passed unmodified. While this sounds pretty straightforward and logical, keep in mind handler methods that take a Message<?> or Object as an argument. By declaring the target type to be Object (which is an instanceof everything in Java), you essentially forfeit the conversion process.

|  |
| --- |
| [Note] |
| Do not expect Message to be converted into some other type based only on the contentType. Remember that the contentType is complementary to the target type. If you wish, you can provide a hint, which MessageConverter may or may not take into consideration. |

**30.1.2 Message Converters**

MessageConverters define two methods:

Object fromMessage(Message<?> message, Class<?> targetClass);

Message<?> toMessage(Object payload, *@Nullable* MessageHeaders headers);

It is important to understand the contract of these methods and their usage, specifically in the context of Spring Cloud Stream.

The fromMessage method converts an incoming Message to an argument type. The payload of the Message could be any type, and it is up to the actual implementation of the MessageConverter to support multiple types. For example, some JSON converter may support the payload type as byte[], String, and others. This is important when the application contains an internal pipeline (that is, input → handler1 → handler2 →. . . → output) and the output of the upstream handler results in a Message which may not be in the initial wire format.

However, the toMessage method has a more strict contract and must always convert Message to the wire format: byte[].

So, for all intents and purposes (and especially when implementing your own converter) you regard the two methods as having the following signatures:

Object fromMessage(Message<?> message, Class<?> targetClass);

Message<**byte**[]> toMessage(Object payload, *@Nullable* MessageHeaders headers);

**30.2 Provided MessageConverters**

As mentioned earlier, the framework already provides a stack of MessageConverters to handle most common use cases. The following list describes the provided MessageConverters, in order of precedence (the first MessageConverter that works is used):

1. ApplicationJsonMessageMarshallingConverter: Variation of the org.springframework.messaging.converter.MappingJackson2MessageConverter. Supports conversion of the payload of the Message to/from POJO for cases when contentType is application/json (DEFAULT).
2. TupleJsonMessageConverter: **DEPRECATED** Supports conversion of the payload of the Message to/from org.springframework.tuple.Tuple.
3. ByteArrayMessageConverter: Supports conversion of the payload of the Message from byte[] to byte[] for cases when contentType is application/octet-stream. It is essentially a pass through and exists primarily for backward compatibility.
4. ObjectStringMessageConverter: Supports conversion of any type to a String when contentType is text/plain. It invokes Object’s toString() method or, if the payload is byte[], a new String(byte[]).
5. JavaSerializationMessageConverter: **DEPRECATED** Supports conversion based on java serialization when contentType is application/x-java-serialized-object.
6. KryoMessageConverter: **DEPRECATED** Supports conversion based on Kryo serialization when contentType is application/x-java-object.
7. JsonUnmarshallingConverter: Similar to the ApplicationJsonMessageMarshallingConverter. It supports conversion of any type when contentType is application/x-java-object. It expects the actual type information to be embedded in the contentType as an attribute (for example, application/x-java-object;type=foo.bar.Cat).

When no appropriate converter is found, the framework throws an exception. When that happens, you should check your code and configuration and ensure you did not miss anything (that is, ensure that you provided a contentType by using a binding or a header). However, most likely, you found some uncommon case (such as a custom contentType perhaps) and the current stack of provided MessageConverters does not know how to convert. If that is the case, you can add custom MessageConverter. See [Section 30.3, “User-defined Message Converters”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-user-defined-message-converters).

**30.3 User-defined Message Converters**

Spring Cloud Stream exposes a mechanism to define and register additional MessageConverters. To use it, implement org.springframework.messaging.converter.MessageConverter, configure it as a @Bean, and annotate it with @StreamMessageConverter. It is then apended to the existing stack of `MessageConverter`s.

|  |
| --- |
| [Note] |
| It is important to understand that custom MessageConverter implementations are added to the head of the existing stack. Consequently, custom MessageConverter implementations take precedence over the existing ones, which lets you override as well as add to the existing converters. |

The following example shows how to create a message converter bean to support a new content type called application/bar:

*@EnableBinding(Sink.class)*

*@SpringBootApplication*

**public** **static** **class** SinkApplication {

...

*@Bean*

*@StreamMessageConverter*

**public** MessageConverter customMessageConverter() {

**return** **new** MyCustomMessageConverter();

}

}

**public** **class** MyCustomMessageConverter **extends** AbstractMessageConverter {

**public** MyCustomMessageConverter() {

**super**(**new** MimeType("application", "bar"));

}

*@Override*

**protected** **boolean** supports(Class<?> clazz) {

**return** (Bar.**class**.equals(clazz));

}

*@Override*

**protected** Object convertFromInternal(Message<?> message, Class<?> targetClass, Object conversionHint) {

Object payload = message.getPayload();

**return** (payload **instanceof** Bar ? payload : **new** Bar((**byte**[]) payload));

}

}

Spring Cloud Stream also provides support for Avro-based converters and schema evolution. See “[Chapter 31, *Schema Evolution Support*](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#schema-evolution)” for details.

**31. Schema Evolution Support**

Spring Cloud Stream provides support for schema evolution so that the data can be evolved over time and still work with older or newer producers and consumers and vice versa. Most serialization models, especially the ones that aim for portability across different platforms and languages, rely on a schema that describes how the data is serialized in the binary payload. In order to serialize the data and then to interpret it, both the sending and receiving sides must have access to a schema that describes the binary format. In certain cases, the schema can be inferred from the payload type on serialization or from the target type on deserialization. However, many applications benefit from having access to an explicit schema that describes the binary data format. A schema registry lets you store schema information in a textual format (typically JSON) and makes that information accessible to various applications that need it to receive and send data in binary format. A schema is referenceable as a tuple consisting of:

* A subject that is the logical name of the schema
* The schema version
* The schema format, which describes the binary format of the data

This following sections goes through the details of various components involved in schema evolution process.

**31.1 Schema Registry Client**

The client-side abstraction for interacting with schema registry servers is the SchemaRegistryClient interface, which has the following structure:

**public** **interface** SchemaRegistryClient {

SchemaRegistrationResponse register(String subject, String format, String schema);

String fetch(SchemaReference schemaReference);

String fetch(Integer id);

}

Spring Cloud Stream provides out-of-the-box implementations for interacting with its own schema server and for interacting with the Confluent Schema Registry.

A client for the Spring Cloud Stream schema registry can be configured by using the @EnableSchemaRegistryClient, as follows:

*@EnableBinding(Sink.class)*

*@SpringBootApplication*

*@EnableSchemaRegistryClient*

**public** **static** **class** AvroSinkApplication {

...

}

|  |
| --- |
| [Note] |
| The default converter is optimized to cache not only the schemas from the remote server but also the parse() and toString() methods, which are quite expensive. Because of this, it uses a DefaultSchemaRegistryClient that does not cache responses. If you intend to change the default behavior, you can use the client directly on your code and override it to the desired outcome. To do so, you have to add the property spring.cloud.stream.schemaRegistryClient.cached=true to your application properties. |

**31.1.1 Schema Registry Client Properties**

The Schema Registry Client supports the following properties:

spring.cloud.stream.schemaRegistryClient.endpoint

The location of the schema-server. When setting this, use a full URL, including protocol (http or https) , port, and context path.

Default

<http://localhost:8990/>

spring.cloud.stream.schemaRegistryClient.cached

Whether the client should cache schema server responses. Normally set to false, as the caching happens in the message converter. Clients using the schema registry client should set this to true.

Default

true

**31.2 Avro Schema Registry Client Message Converters**

For applications that have a SchemaRegistryClient bean registered with the application context, Spring Cloud Stream auto configures an Apache Avro message converter for schema management. This eases schema evolution, as applications that receive messages can get easy access to a writer schema that can be reconciled with their own reader schema.

For outbound messages, if the content type of the channel is set to application/\*+avro, the MessageConverter is activated, as shown in the following example:

spring.cloud.stream.bindings.output.contentType=application/\*+avro

During the outbound conversion, the message converter tries to infer the schema of each outbound messages (based on its type) and register it to a subject (based on the payload type) by using the SchemaRegistryClient. If an identical schema is already found, then a reference to it is retrieved. If not, the schema is registered, and a new version number is provided. The message is sent with a contentType header by using the following scheme: application/[prefix].[subject].v[version]+avro, where prefix is configurable and subject is deduced from the payload type.

For example, a message of the type User might be sent as a binary payload with a content type of application/vnd.user.v2+avro, where user is the subject and 2 is the version number.

When receiving messages, the converter infers the schema reference from the header of the incoming message and tries to retrieve it. The schema is used as the writer schema in the deserialization process.

**31.2.1 Avro Schema Registry Message Converter Properties**

If you have enabled Avro based schema registry client by setting spring.cloud.stream.bindings.output.contentType=application/\*+avro, you can customize the behavior of the registration by setting the following properties.

spring.cloud.stream.schema.avro.dynamicSchemaGenerationEnabled

Enable if you want the converter to use reflection to infer a Schema from a POJO.

Default: false

spring.cloud.stream.schema.avro.readerSchema

Avro compares schema versions by looking at a writer schema (origin payload) and a reader schema (your application payload). See the [Avro documentation](https://avro.apache.org/docs/1.7.6/spec.html) for more information. If set, this overrides any lookups at the schema server and uses the local schema as the reader schema. Default: null

spring.cloud.stream.schema.avro.schemaLocations

Registers any .avsc files listed in this property with the Schema Server.

Default: empty

spring.cloud.stream.schema.avro.prefix

The prefix to be used on the Content-Type header.

Default: vnd

**31.3 Apache Avro Message Converters**

Spring Cloud Stream provides support for schema-based message converters through its spring-cloud-stream-schema module. Currently, the only serialization format supported out of the box for schema-based message converters is Apache Avro, with more formats to be added in future versions.

The spring-cloud-stream-schema module contains two types of message converters that can be used for Apache Avro serialization:

* Converters that use the class information of the serialized or deserialized objects or a schema with a location known at startup.
* Converters that use a schema registry. They locate the schemas at runtime and dynamically register new schemas as domain objects evolve.

**31.4 Converters with Schema Support**

The AvroSchemaMessageConverter supports serializing and deserializing messages either by using a predefined schema or by using the schema information available in the class (either reflectively or contained in the SpecificRecord). If you provide a custom converter, then the default AvroSchemaMessageConverter bean is not created. The following example shows a custom converter:

To use custom converters, you can simply add it to the application context, optionally specifying one or more MimeTypes with which to associate it. The default MimeType is application/avro.

If the target type of the conversion is a GenericRecord, a schema must be set.

The following example shows how to configure a converter in a sink application by registering the Apache Avro MessageConverter without a predefined schema. In this example, note that the mime type value is avro/bytes, not the default application/avro.

*@EnableBinding(Sink.class)*

*@SpringBootApplication*

**public** **static** **class** SinkApplication {

...

*@Bean*

**public** MessageConverter userMessageConverter() {

**return** **new** AvroSchemaMessageConverter(MimeType.valueOf("avro/bytes"));

}

}

Conversely, the following application registers a converter with a predefined schema (found on the classpath):

*@EnableBinding(Sink.class)*

*@SpringBootApplication*

**public** **static** **class** SinkApplication {

...

*@Bean*

**public** MessageConverter userMessageConverter() {

AvroSchemaMessageConverter converter = **new** AvroSchemaMessageConverter(MimeType.valueOf("avro/bytes"));

converter.setSchemaLocation(**new** ClassPathResource("schemas/User.avro"));

**return** converter;

}

}

**31.5 Schema Registry Server**

Spring Cloud Stream provides a schema registry server implementation. To use it, you can add the spring-cloud-stream-schema-server artifact to your project and use the @EnableSchemaRegistryServer annotation, which adds the schema registry server REST controller to your application. This annotation is intended to be used with Spring Boot web applications, and the listening port of the server is controlled by the server.port property. The spring.cloud.stream.schema.server.pathproperty can be used to control the root path of the schema server (especially when it is embedded in other applications). The spring.cloud.stream.schema.server.allowSchemaDeletion boolean property enables the deletion of a schema. By default, this is disabled.

The schema registry server uses a relational database to store the schemas. By default, it uses an embedded database. You can customize the schema storage by using the [Spring Boot SQL database and JDBC configuration options](https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#boot-features-sql).

The following example shows a Spring Boot application that enables the schema registry:

*@SpringBootApplication*

*@EnableSchemaRegistryServer*

**public** **class** SchemaRegistryServerApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.run(SchemaRegistryServerApplication.**class**, args);

}

}

**31.5.1 Schema Registry Server API**

The Schema Registry Server API consists of the following operations:

* POST / — see “[the section called “Registering a New Schema”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-registering-new-schema)”
* 'GET /{subject}/{format}/{version}' — see “[the section called “Retrieving an Existing Schema by Subject, Format, and Version”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-retrieve-schema-subject-format-version)”
* GET /{subject}/{format} — see “[the section called “Retrieving an Existing Schema by Subject and Format”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-retrieve-schema-subject-format)”
* GET /schemas/{id} — see “[the section called “Retrieving an Existing Schema by ID”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-retrieve-schema-id)”
* DELETE /{subject}/{format}/{version} — see “[the section called “Deleting a Schema by Subject, Format, and Version”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-deleting-schema-subject-format-version)”
* DELETE /schemas/{id} — see “[the section called “Deleting a Schema by ID”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-deleting-schema-id)”
* DELETE /{subject} — see “[the section called “Deleting a Schema by Subject”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-deleting-schema-subject)”

**Registering a New Schema**

To register a new schema, send a POST request to the / endpoint.

The / accepts a JSON payload with the following fields:

* subject: The schema subject
* format: The schema format
* definition: The schema definition

Its response is a schema object in JSON, with the following fields:

* id: The schema ID
* subject: The schema subject
* format: The schema format
* version: The schema version
* definition: The schema definition

**Retrieving an Existing Schema by Subject, Format, and Version**

To retrieve an existing schema by subject, format, and version, send GET request to the /{subject}/{format}/{version} endpoint.

Its response is a schema object in JSON, with the following fields:

* id: The schema ID
* subject: The schema subject
* format: The schema format
* version: The schema version
* definition: The schema definition

**Retrieving an Existing Schema by Subject and Format**

To retrieve an existing schema by subject and format, send a GET request to the /subject/format endpoint.

Its response is a list of schemas with each schema object in JSON, with the following fields:

* id: The schema ID
* subject: The schema subject
* format: The schema format
* version: The schema version
* definition: The schema definition

**Retrieving an Existing Schema by ID**

To retrieve a schema by its ID, send a GET request to the /schemas/{id} endpoint.

Its response is a schema object in JSON, with the following fields:

* id: The schema ID
* subject: The schema subject
* format: The schema format
* version: The schema version
* definition: The schema definition

**Deleting a Schema by Subject, Format, and Version**

To delete a schema identified by its subject, format, and version, send a DELETE request to the /{subject}/{format}/{version} endpoint.

**Deleting a Schema by ID**

To delete a schema by its ID, send a DELETE request to the /schemas/{id} endpoint.

**Deleting a Schema by Subject**

DELETE /{subject}

Delete existing schemas by their subject.

|  |
| --- |
| [Note] |
| This note applies to users of Spring Cloud Stream 1.1.0.RELEASE only. Spring Cloud Stream 1.1.0.RELEASE used the table name, schema, for storing Schema objects. Schema is a keyword in a number of database implementations. To avoid any conflicts in the future, starting with 1.1.1.RELEASE, we have opted for the name SCHEMA\_REPOSITORY for the storage table. Any Spring Cloud Stream 1.1.0.RELEASE users who upgrade should migrate their existing schemas to the new table before upgrading. |

**31.5.2 Using Confluent’s Schema Registry**

The default configuration creates a DefaultSchemaRegistryClient bean. If you want to use the Confluent schema registry, you need to create a bean of type ConfluentSchemaRegistryClient, which supersedes the one configured by default by the framework. The following example shows how to create such a bean:

*@Bean*

**public** SchemaRegistryClient schemaRegistryClient(*@Value("${spring.cloud.stream.schemaRegistryClient.endpoint}")* String endpoint){

ConfluentSchemaRegistryClient client = **new** ConfluentSchemaRegistryClient();

client.setEndpoint(endpoint);

**return** client;

}

|  |
| --- |
| [Note] |
| The ConfluentSchemaRegistryClient is tested against Confluent platform version 4.0.0. |

**31.6 Schema Registration and Resolution**

To better understand how Spring Cloud Stream registers and resolves new schemas and its use of Avro schema comparison features, we provide two separate subsections:

* “[Section 31.6.1, “Schema Registration Process (Serialization)”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-schema-registration-process)”
* “[Section 31.6.2, “Schema Resolution Process (Deserialization)”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-schema-resolution-process)”

**31.6.1 Schema Registration Process (Serialization)**

The first part of the registration process is extracting a schema from the payload that is being sent over a channel. Avro types such as SpecificRecord or GenericRecord already contain a schema, which can be retrieved immediately from the instance. In the case of POJOs, a schema is inferred if the spring.cloud.stream.schema.avro.dynamicSchemaGenerationEnabled property is set to true (the default).

**Figure 31.1. Schema Writer Resolution Process**

Ones a schema is obtained, the converter loads its metadata (version) from the remote server. First, it queries a local cache. If no result is found, it submits the data to the server, which replies with versioning information. The converter always caches the results to avoid the overhead of querying the Schema Server for every new message that needs to be serialized.

**Figure 31.2. Schema Registration Process**

With the schema version information, the converter sets the contentType header of the message to carry the version information — for example: application/vnd.user.v1+avro.

**31.6.2 Schema Resolution Process (Deserialization)**

When reading messages that contain version information (that is, a contentType header with a scheme like the one described under “[Section 31.6.1, “Schema Registration Process (Serialization)”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-schema-registration-process)”), the converter queries the Schema server to fetch the writer schema of the message. Once it has found the correct schema of the incoming message, it retrieves the reader schema and, by using Avro’s schema resolution support, reads it into the reader definition (setting defaults and any missing properties).

**Figure 31.3. Schema Reading Resolution Process**

|  |
| --- |
| [Note] |
| You should understand the difference between a writer schema (the application that wrote the message) and a reader schema (the receiving application). We suggest taking a moment to read [the Avro terminology](https://avro.apache.org/docs/1.7.6/spec.html) and understand the process. Spring Cloud Stream always fetches the writer schema to determine how to read a message. If you want to get Avro’s schema evolution support working, you need to make sure that a readerSchema was properly set for your application. |

**32. Inter-Application Communication**

Spring Cloud Stream enables communication between applications. Inter-application communication is a complex issue spanning several concerns, as described in the following topics:

* “[Section 32.1, “Connecting Multiple Application Instances”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-connecting-multiple-application-instances)”
* “[Section 32.2, “Instance Index and Instance Count”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-instance-index-instance-count)”
* “[Section 32.3, “Partitioning”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-partitioning)”

**32.1 Connecting Multiple Application Instances**

While Spring Cloud Stream makes it easy for individual Spring Boot applications to connect to messaging systems, the typical scenario for Spring Cloud Stream is the creation of multi-application pipelines, where microservice applications send data to each other. You can achieve this scenario by correlating the input and output destinations of “adjacent” applications.

Suppose a design calls for the Time Source application to send data to the Log Sink application. You could use a common destination named ticktock for bindings within both applications.

Time Source (that has the channel name output) would set the following property:

spring.cloud.stream.bindings.output.destination=ticktock

Log Sink (that has the channel name input) would set the following property:

spring.cloud.stream.bindings.input.destination=ticktock

**32.2 Instance Index and Instance Count**

When scaling up Spring Cloud Stream applications, each instance can receive information about how many other instances of the same application exist and what its own instance index is. Spring Cloud Stream does this through the spring.cloud.stream.instanceCount and spring.cloud.stream.instanceIndex properties. For example, if there are three instances of a HDFS sink application, all three instances have spring.cloud.stream.instanceCount set to 3, and the individual applications have spring.cloud.stream.instanceIndex set to 0, 1, and 2, respectively.

When Spring Cloud Stream applications are deployed through Spring Cloud Data Flow, these properties are configured automatically; when Spring Cloud Stream applications are launched independently, these properties must be set correctly. By default, spring.cloud.stream.instanceCount is 1, and spring.cloud.stream.instanceIndex is 0.

In a scaled-up scenario, correct configuration of these two properties is important for addressing partitioning behavior (see below) in general, and the two properties are always required by certain binders (for example, the Kafka binder) in order to ensure that data are split correctly across multiple consumer instances.

**32.3 Partitioning**

Partitioning in Spring Cloud Stream consists of two tasks:

* “[Section 32.3.1, “Configuring Output Bindings for Partitioning”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-configuring-output-bindings-partitioning)”
* “[Section 32.3.2, “Configuring Input Bindings for Partitioning”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-configuring-input-bindings-partitioning)”

**32.3.1 Configuring Output Bindings for Partitioning**

You can configure an output binding to send partitioned data by setting one and only one of its partitionKeyExpression or partitionKeyExtractorNameproperties, as well as its partitionCount property.

For example, the following is a valid and typical configuration:

spring.cloud.stream.bindings.output.producer.partitionKeyExpression=payload.id

spring.cloud.stream.bindings.output.producer.partitionCount=5

Based on that example configuration, data is sent to the target partition by using the following logic.

A partition key’s value is calculated for each message sent to a partitioned output channel based on the partitionKeyExpression. The partitionKeyExpression is a SpEL expression that is evaluated against the outbound message for extracting the partitioning key.

If a SpEL expression is not sufficient for your needs, you can instead calculate the partition key value by providing an implementation of org.springframework.cloud.stream.binder.PartitionKeyExtractorStrategy and configuring it as a bean (by using the @Bean annotation). If you have more then one bean of type org.springframework.cloud.stream.binder.PartitionKeyExtractorStrategy available in the Application Context, you can further filter it by specifying its name with the partitionKeyExtractorName property, as shown in the following example:

--spring.cloud.stream.bindings.output.producer.partitionKeyExtractorName=customPartitionKeyExtractor

--spring.cloud.stream.bindings.output.producer.partitionCount=5

. . .

@Bean

public CustomPartitionKeyExtractorClass customPartitionKeyExtractor() {

return new CustomPartitionKeyExtractorClass();

}

|  |
| --- |
| [Note] |
| In previous versions of Spring Cloud Stream, you could specify the implementation of org.springframework.cloud.stream.binder.PartitionKeyExtractorStrategy by setting the spring.cloud.stream.bindings.output.producer.partitionKeyExtractorClass property. Since version 2.0, this property is deprecated, and support for it will be removed in a future version. |

Once the message key is calculated, the partition selection process determines the target partition as a value between 0 and partitionCount - 1. The default calculation, applicable in most scenarios, is based on the following formula: key.hashCode() % partitionCount. This can be customized on the binding, either by setting a SpEL expression to be evaluated against the 'key' (through the partitionSelectorExpression property) or by configuring an implementation of org.springframework.cloud.stream.binder.PartitionSelectorStrategy as a bean (by using the @Bean annotation). Similar to the PartitionKeyExtractorStrategy, you can further filter it by using the spring.cloud.stream.bindings.output.producer.partitionSelectorName property when more than one bean of this type is available in the Application Context, as shown in the following example:

--spring.cloud.stream.bindings.output.producer.partitionSelectorName=customPartitionSelector

. . .

@Bean

public CustomPartitionSelectorClass customPartitionSelector() {

return new CustomPartitionSelectorClass();

}

|  |
| --- |
| [Note] |
| In previous versions of Spring Cloud Stream you could specify the implementation of org.springframework.cloud.stream.binder.PartitionSelectorStrategy by setting the spring.cloud.stream.bindings.output.producer.partitionSelectorClass property. Since version 2.0, this property is deprecated and support for it will be removed in a future version. |

**32.3.2 Configuring Input Bindings for Partitioning**

An input binding (with the channel name input) is configured to receive partitioned data by setting its partitioned property, as well as the instanceIndex and instanceCount properties on the application itself, as shown in the following example:

spring.cloud.stream.bindings.input.consumer.partitioned=true

spring.cloud.stream.instanceIndex=3

spring.cloud.stream.instanceCount=5

The instanceCount value represents the total number of application instances between which the data should be partitioned. The instanceIndex must be a unique value across the multiple instances, with a value between 0 and instanceCount - 1. The instance index helps each application instance to identify the unique partition(s) from which it receives data. It is required by binders using technology that does not support partitioning natively. For example, with RabbitMQ, there is a queue for each partition, with the queue name containing the instance index. With Kafka, if autoRebalanceEnabled is true (default), Kafka takes care of distributing partitions across instances, and these properties are not required. If autoRebalanceEnabled is set to false, the instanceCount and instanceIndex are used by the binder to determine which partition(s) the instance subscribes to (you must have at least as many partitions as there are instances). The binder allocates the partitions instead of Kafka. This might be useful if you want messages for a particular partition to always go to the same instance. When a binder configuration requires them, it is important to set both values correctly in order to ensure that all of the data is consumed and that the application instances receive mutually exclusive datasets.

While a scenario in which using multiple instances for partitioned data processing may be complex to set up in a standalone case, Spring Cloud Dataflow can simplify the process significantly by populating both the input and output values correctly and by letting you rely on the runtime infrastructure to provide information about the instance index and instance count.

**33. Testing**

Spring Cloud Stream provides support for testing your microservice applications without connecting to a messaging system. You can do that by using the TestSupportBinder provided by the spring-cloud-stream-test-support library, which can be added as a test dependency to the application, as shown in the following example:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-stream-test-support</artifactId>

<scope>test</scope>

</dependency>

|  |
| --- |
| [Note] |
| The TestSupportBinder uses the Spring Boot autoconfiguration mechanism to supersede the other binders found on the classpath. Therefore, when adding a binder as a dependency, you must make sure that the test scope is being used. |

The TestSupportBinder lets you interact with the bound channels and inspect any messages sent and received by the application.

For outbound message channels, the TestSupportBinder registers a single subscriber and retains the messages emitted by the application in a MessageCollector. They can be retrieved during tests and have assertions made against them.

You can also send messages to inbound message channels so that the consumer application can consume the messages. The following example shows how to test both input and output channels on a processor:

*@RunWith(SpringRunner.class)*

*@SpringBootTest(webEnvironment= SpringBootTest.WebEnvironment.RANDOM\_PORT)*

**public** **class** ExampleTest {

*@Autowired*

**private** Processor processor;

*@Autowired*

**private** MessageCollector messageCollector;

*@Test*

*@SuppressWarnings("unchecked")*

**public** **void** testWiring() {

Message<String> message = **new** GenericMessage<>("hello");

processor.input().send(message);

Message<String> received = (Message<String>) messageCollector.forChannel(processor.output()).poll();

assertThat(received.getPayload(), equalTo("hello world"));

}

*@SpringBootApplication*

*@EnableBinding(Processor.class)*

**public** **static** **class** MyProcessor {

*@Autowired*

**private** Processor channels;

*@Transformer(inputChannel = Processor.INPUT, outputChannel = Processor.OUTPUT)*

**public** String transform(String in) {

**return** in + " world";

}

}

}

In the preceding example, we create an application that has an input channel and an output channel, both bound through the Processor interface. The bound interface is injected into the test so that we can have access to both channels. We send a message on the input channel, and we use the MessageCollector provided by Spring Cloud Stream’s test support to capture that the message has been sent to the output channel as a result. Once we have received the message, we can validate that the component functions correctly.

**33.1 Disabling the Test Binder Autoconfiguration**

The intent behind the test binder superseding all the other binders on the classpath is to make it easy to test your applications without making changes to your production dependencies. In some cases (for example, integration tests) it is useful to use the actual production binders instead, and that requires disabling the test binder autoconfiguration. To do so, you can exclude the org.springframework.cloud.stream.test.binder.TestSupportBinderAutoConfiguration class by using one of the Spring Boot autoconfiguration exclusion mechanisms, as shown in the following example:

*@SpringBootApplication(exclude = TestSupportBinderAutoConfiguration.class)*

*@EnableBinding(Processor.class)*

**public** **static** **class** MyProcessor {

*@Transformer(inputChannel = Processor.INPUT, outputChannel = Processor.OUTPUT)*

**public** String transform(String in) {

**return** in + " world";

}

}

When autoconfiguration is disabled, the test binder is available on the classpath, and its defaultCandidate property is set to false so that it does not interfere with the regular user configuration. It can be referenced under the name, test, as shown in the following example:

spring.cloud.stream.defaultBinder=test

**34. Health Indicator**

Spring Cloud Stream provides a health indicator for binders. It is registered under the name binders and can be enabled or disabled by setting the management.health.binders.enabled property.

By default management.health.binders.enabled is set to false. Setting management.health.binders.enabled to true enables the health indicator, allowing you to access the /health endpoint to retrieve the binder health indicators.

Health indicators are binder-specific and certain binder implementations may not necessarily provide a health indicator.

**35. Metrics Emitter**

Spring Boot Actuator provides dependency management and auto-configuration for [Micrometer](https://micrometer.io/), an application metrics facade that supports numerous [monitoring systems](https://docs.spring.io/spring-boot/docs/2.0.0.RELEASE/reference/htmlsingle/#production-ready-metrics).

Spring Cloud Stream provides support for emitting any available micrometer-based metrics to a binding destination, allowing for periodic collection of metric data from stream applications without relying on polling individual endpoints.

Metrics Emitter is activated by defining the spring.cloud.stream.bindings.applicationMetrics.destination property, which specifies the name of the binding destination used by the current binder to publish metric messages.

For example:

spring.cloud.stream.bindings.applicationMetrics.destination=myMetricDestination

The preceding example instructs the binder to bind to myMetricDestination (that is, Rabbit exchange, Kafka topic, and others).

The following properties can be used for customizing the emission of metrics:

spring.cloud.stream.metrics.key

The name of the metric being emitted. Should be a unique value per application.

Default: ${spring.application.name:${vcap.application.name:${spring.config.name:application}}}

spring.cloud.stream.metrics.properties

Allows white listing application properties that are added to the metrics payload

Default: null.

spring.cloud.stream.metrics.meter-filter

Pattern to control the 'meters' one wants to capture. For example, specifying spring.integration.\* captures metric information for meters whose name starts with spring.integration.

Default: all 'meters' are captured.

spring.cloud.stream.metrics.schedule-interval

Interval to control the rate of publishing metric data.

Default: 1 min

Consider the following:

java -jar time-source.jar \

--spring.cloud.stream.bindings.applicationMetrics.destination=someMetrics \

--spring.cloud.stream.metrics.properties=spring.application\*\* \

--spring.cloud.stream.metrics.meter-filter=spring.integration.\*

The following example shows the payload of the data published to the binding destination as a result of the preceding command:

{

"name": "application",

"createdTime": "2018-03-23T14:48:12.700Z",

"properties": {

},

"metrics": [

{

"id": {

"name": "spring.integration.send",

"tags": [

{

"key": "exception",

"value": "none"

},

{

"key": "name",

"value": "input"

},

{

"key": "result",

"value": "success"

},

{

"key": "type",

"value": "channel"

}

],

"type": "TIMER",

"description": "Send processing time",

"baseUnit": "milliseconds"

},

"timestamp": "2018-03-23T14:48:12.697Z",

"sum": 130.340546,

"count": 6,

"mean": 21.72342433333333,

"upper": 116.176299,

"total": 130.340546

}

]

}

|  |
| --- |
| [Note] |
| Given that the format of the Metric message has slightly changed after migrating to Micrometer, the published message will also have a STREAM\_CLOUD\_STREAM\_VERSION header set to 2.x to help distinguish between Metric messages from the older versions of the Spring Cloud Stream. |

**36. Samples**

For Spring Cloud Stream samples, see the [spring-cloud-stream-samples](https://github.com/spring-cloud/spring-cloud-stream-samples) repository on GitHub.

**36.1 Deploying Stream Applications on CloudFoundry**

On CloudFoundry, services are usually exposed through a special environment variable called [VCAP\_SERVICES](https://docs.cloudfoundry.org/devguide/deploy-apps/environment-variable.html#VCAP-SERVICES).

When configuring your binder connections, you can use the values from an environment variable as explained on the [dataflow Cloud Foundry Server](https://docs.spring.io/spring-cloud-dataflow-server-cloudfoundry/docs/current-SNAPSHOT/reference/htmlsingle/#getting-started-ups) docs.

**Part VI. Binder Implementations**

**37. Apache Kafka Binder**

**37.1 Usage**

To use Apache Kafka binder, you need to add spring-cloud-stream-binder-kafka as a dependency to your Spring Cloud Stream application, as shown in the following example for Maven:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-stream-binder-kafka</artifactId>

</dependency>

Alternatively, you can also use the Spring Cloud Stream Kafka Starter, as shown inn the following example for Maven:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-stream-kafka</artifactId>

</dependency>

**37.2 Apache Kafka Binder Overview**

The following image shows a simplified diagram of how the Apache Kafka binder operates:

**Figure 37.1. Kafka Binder**

The Apache Kafka Binder implementation maps each destination to an Apache Kafka topic. The consumer group maps directly to the same Apache Kafka concept. Partitioning also maps directly to Apache Kafka partitions as well.

The binder currently uses the Apache Kafka kafka-clients 1.0.0 jar and is designed to be used with a broker of at least that version. This client can communicate with older brokers (see the Kafka documentation), but certain features may not be available. For example, with versions earlier than 0.11.x.x, native headers are not supported. Also, 0.11.x.x does not support the autoAddPartitions property.

**37.3 Configuration Options**

This section contains the configuration options used by the Apache Kafka binder.

For common configuration options and properties pertaining to binder, see the [core documentation](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#binding-properties).

**37.3.1 Kafka Binder Properties**

spring.cloud.stream.kafka.binder.brokers

A list of brokers to which the Kafka binder connects.

Default: localhost.

spring.cloud.stream.kafka.binder.defaultBrokerPort

brokers allows hosts specified with or without port information (for example, host1,host2:port2). This sets the default port when no port is configured in the broker list.

Default: 9092.

spring.cloud.stream.kafka.binder.configuration

Key/Value map of client properties (both producers and consumer) passed to all clients created by the binder. Due to the fact that these properties are used by both producers and consumers, usage should be restricted to common properties — for example, security settings.

Default: Empty map.

spring.cloud.stream.kafka.binder.headers

The list of custom headers that are transported by the binder. Only required when communicating with older applications (⇐ 1.3.x) with a kafka-clients version < 0.11.0.0. Newer versions support headers natively.

Default: empty.

spring.cloud.stream.kafka.binder.healthTimeout

The time to wait to get partition information, in seconds. Health reports as down if this timer expires.

Default: 10.

spring.cloud.stream.kafka.binder.requiredAcks

The number of required acks on the broker. See the Kafka documentation for the producer acks property.

Default: 1.

spring.cloud.stream.kafka.binder.minPartitionCount

Effective only if autoCreateTopics or autoAddPartitions is set. The global minimum number of partitions that the binder configures on topics on which it produces or consumes data. It can be superseded by the partitionCount setting of the producer or by the value of instanceCount \* concurrency settings of the producer (if either is larger).

Default: 1.

spring.cloud.stream.kafka.binder.replicationFactor

The replication factor of auto-created topics if autoCreateTopics is active. Can be overridden on each binding.

Default: 1.

spring.cloud.stream.kafka.binder.autoCreateTopics

If set to true, the binder creates new topics automatically. If set to false, the binder relies on the topics being already configured. In the latter case, if the topics do not exist, the binder fails to start.

|  |
| --- |
| [Note] |
| This setting is independent of the auto.topic.create.enable setting of the broker and does not influence it. If the server is set to auto-create topics, they may be created as part of the metadata retrieval request, with default broker settings. |

Default: true.

spring.cloud.stream.kafka.binder.autoAddPartitions

If set to true, the binder creates new partitions if required. If set to false, the binder relies on the partition size of the topic being already configured. If the partition count of the target topic is smaller than the expected value, the binder fails to start.

Default: false.

spring.cloud.stream.kafka.binder.transaction.transactionIdPrefix

Enables transactions in the binder. See transaction.id in the Kafka documentation and [Transactions](https://docs.spring.io/spring-kafka/reference/html/_reference.html#transactions) in the spring-kafka documentation. When transactions are enabled, individual producer properties are ignored and all producers use the spring.cloud.stream.kafka.binder.transaction.producer.\*properties.

Default null (no transactions)

spring.cloud.stream.kafka.binder.transaction.producer.\*

Global producer properties for producers in a transactional binder. See spring.cloud.stream.kafka.binder.transaction.transactionIdPrefix and [Section 37.3.3, “Kafka Producer Properties”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#kafka-producer-properties) and the general producer properties supported by all binders.

Default: See individual producer properties.

spring.cloud.stream.kafka.binder.headerMapperBeanName

The bean name of a KafkaHeaderMapper used for mapping spring-messaging headers to and from Kafka headers. Use this, for example, if you wish to customize the trusted packages in a DefaultKafkaHeaderMapper that uses JSON deserialization for the headers.

Default: none.

**37.3.2 Kafka Consumer Properties**

The following properties are available for Kafka consumers only and must be prefixed with spring.cloud.stream.kafka.bindings.<channelName>.consumer..

admin.configuration

A Map of Kafka topic properties used when provisioning topics — for example, spring.cloud.stream.kafka.bindings.input.consumer.admin.configuration.message.format.version=0.9.0.0

Default: none.

admin.replicas-assignment

A Map<Integer, List<Integer>> of replica assignments, with the key being the partition and the value being the assignments. Used when provisioning new topics. See the NewTopic Javadocs in the kafka-clients jar.

Default: none.

admin.replication-factor

The replication factor to use when provisioning topics. Overrides the binder-wide setting. Ignored if replicas-assignments is present.

Default: none (the binder-wide default of 1 is used).

autoRebalanceEnabled

When true, topic partitions is automatically rebalanced between the members of a consumer group. When false, each consumer is assigned a fixed set of partitions based on spring.cloud.stream.instanceCount and spring.cloud.stream.instanceIndex. This requires both the spring.cloud.stream.instanceCount and spring.cloud.stream.instanceIndex properties to be set appropriately on each launched instance. The value of the spring.cloud.stream.instanceCount property must typically be greater than 1 in this case.

Default: true.

ackEachRecord

When autoCommitOffset is true, this setting dictates whether to commit the offset after each record is processed. By default, offsets are committed after all records in the batch of records returned by consumer.poll() have been processed. The number of records returned by a poll can be controlled with the max.poll.records Kafka property, which is set through the consumer configuration property. Setting this to true may cause a degradation in performance, but doing so reduces the likelihood of redelivered records when a failure occurs. Also, see the binder requiredAcks property, which also affects the performance of committing offsets.

Default: false.

autoCommitOffset

Whether to autocommit offsets when a message has been processed. If set to false, a header with the key kafka\_acknowledgment of the type org.springframework.kafka.support.Acknowledgment header is present in the inbound message. Applications may use this header for acknowledging messages. See the examples section for details. When this property is set to false, Kafka binder sets the ack mode to org.springframework.kafka.listener.AbstractMessageListenerContainer.AckMode.MANUAL and the application is responsible for acknowledging records. Also see ackEachRecord.

Default: true.

autoCommitOnError

Effective only if autoCommitOffset is set to true. If set to false, it suppresses auto-commits for messages that result in errors and commits only for successful messages. It allows a stream to automatically replay from the last successfully processed message, in case of persistent failures. If set to true, it always auto-commits (if auto-commit is enabled). If not set (the default), it effectively has the same value as enableDlq, auto-committing erroneous messages if they are sent to a DLQ and not committing them otherwise.

Default: not set.

resetOffsets

Whether to reset offsets on the consumer to the value provided by startOffset.

Default: false.

startOffset

The starting offset for new groups. Allowed values: earliest and latest. If the consumer group is set explicitly for the consumer 'binding' (through spring.cloud.stream.bindings.<channelName>.group), 'startOffset' is set to earliest. Otherwise, it is set to latest for the anonymous consumer group. Also see resetOffsets (earlier in this list).

Default: null (equivalent to earliest).

enableDlq

When set to true, it enables DLQ behavior for the consumer. By default, messages that result in errors are forwarded to a topic named error.<destination>.<group>. The DLQ topic name can be configurable by setting the dlqName property. This provides an alternative option to the more common Kafka replay scenario for the case when the number of errors is relatively small and replaying the entire original topic may be too cumbersome. See [Section 37.6, “Dead-Letter Topic Processing”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#kafka-dlq-processing) processing for more information. Starting with version 2.0, messages sent to the DLQ topic are enhanced with the following headers: x-original-topic, x-exception-message, and x-exception-stacktrace as byte[].

Default: false.

configuration

Map with a key/value pair containing generic Kafka consumer properties.

Default: Empty map.

dlqName

The name of the DLQ topic to receive the error messages.

Default: null (If not specified, messages that result in errors are forwarded to a topic named error.<destination>.<group>).

dlqProducerProperties

Using this, DLQ-specific producer properties can be set. All the properties available through kafka producer properties can be set through this property.

Default: Default Kafka producer properties.

standardHeaders

Indicates which standard headers are populated by the inbound channel adapter. Allowed values: none, id, timestamp, or both. Useful if using native deserialization and the first component to receive a message needs an id (such as an aggregator that is configured to use a JDBC message store).

Default: none

converterBeanName

The name of a bean that implements RecordMessageConverter. Used in the inbound channel adapter to replace the default MessagingMessageConverter.

Default: null

idleEventInterval

The interval, in milliseconds, between events indicating that no messages have recently been received. Use an ApplicationListener<ListenerContainerIdleEvent> to receive these events. See [the section called “Example: Pausing and Resuming the Consumer”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#pause-resume) for a usage example.

Default: 30000

**37.3.3 Kafka Producer Properties**

The following properties are available for Kafka producers only and must be prefixed with spring.cloud.stream.kafka.bindings.<channelName>.producer..

admin.configuration

A Map of Kafka topic properties used when provisioning new topics — for example, spring.cloud.stream.kafka.bindings.input.consumer.admin.configuration.message.format.version=0.9.0.0

Default: none.

admin.replicas-assignment

A Map<Integer, List<Integer>> of replica assignments, with the key being the partition and the value being the assignments. Used when provisioning new topics. See NewTopic javadocs in the kafka-clients jar.

Default: none.

admin.replication-factor

The replication factor to use when provisioning new topics. Overrides the binder-wide setting. Ignored if replicas-assignments is present.

Default: none (the binder-wide default of 1 is used).

bufferSize

Upper limit, in bytes, of how much data the Kafka producer attempts to batch before sending.

Default: 16384.

sync

Whether the producer is synchronous.

Default: false.

batchTimeout

How long the producer waits to allow more messages to accumulate in the same batch before sending the messages. (Normally, the producer does not wait at all and simply sends all the messages that accumulated while the previous send was in progress.) A non-zero value may increase throughput at the expense of latency.

Default: 0.

messageKeyExpression

A SpEL expression evaluated against the outgoing message used to populate the key of the produced Kafka message — for example, headers['myKey']. The payload cannot be used because, by the time this expression is evaluated, the payload is already in the form of a byte[].

Default: none.

headerPatterns

A comma-delimited list of simple patterns to match Spring messaging headers to be mapped to the Kafka Headers in the ProducerRecord. Patterns can begin or end with the wildcard character (asterisk). Patterns can be negated by prefixing with !. Matching stops after the first match (positive or negative). For example !ask,as\* will pass ash but not ask. id and timestamp are never mapped.

Default: \* (all headers - except the id and timestamp)

configuration

Map with a key/value pair containing generic Kafka producer properties.

Default: Empty map.

|  |
| --- |
| [Note] |
| The Kafka binder uses the partitionCount setting of the producer as a hint to create a topic with the given partition count (in conjunction with the minPartitionCount, the maximum of the two being the value being used). Exercise caution when configuring both minPartitionCount for a binder and partitionCount for an application, as the larger value is used. If a topic already exists with a smaller partition count and autoAddPartitions is disabled (the default), the binder fails to start. If a topic already exists with a smaller partition count and autoAddPartitions is enabled, new partitions are added. If a topic already exists with a larger number of partitions than the maximum of (minPartitionCount or partitionCount), the existing partition count is used. |

**37.3.4 Usage examples**

In this section, we show the use of the preceding properties for specific scenarios.

**Example: Setting autoCommitOffset to false and Relying on Manual Acking**

This example illustrates how one may manually acknowledge offsets in a consumer application.

This example requires that spring.cloud.stream.kafka.bindings.input.consumer.autoCommitOffset be set to false. Use the corresponding input channel name for your example.

@SpringBootApplication

@EnableBinding(Sink.class)

public class ManuallyAcknowdledgingConsumer {

public static void main(String[] args) {

SpringApplication.run(ManuallyAcknowdledgingConsumer.class, args);

}

@StreamListener(Sink.INPUT)

public void process(Message<?> message) {

Acknowledgment acknowledgment = message.getHeaders().get(KafkaHeaders.ACKNOWLEDGMENT, Acknowledgment.class);

if (acknowledgment != null) {

System.out.println("Acknowledgment provided");

acknowledgment.acknowledge();

}

}

}

**Example: Security Configuration**

Apache Kafka 0.9 supports secure connections between client and brokers. To take advantage of this feature, follow the guidelines in the [Apache Kafka Documentation](https://kafka.apache.org/090/documentation.html#security_configclients)as well as the Kafka 0.9 [security guidelines from the Confluent documentation](http://docs.confluent.io/2.0.0/kafka/security.html). Use the spring.cloud.stream.kafka.binder.configuration option to set security properties for all clients created by the binder.

For example, to set security.protocol to SASL\_SSL, set the following property:

spring.cloud.stream.kafka.binder.configuration.security.protocol=SASL\_SSL

All the other security properties can be set in a similar manner.

When using Kerberos, follow the instructions in the [reference documentation](https://kafka.apache.org/090/documentation.html#security_sasl_clientconfig) for creating and referencing the JAAS configuration.

Spring Cloud Stream supports passing JAAS configuration information to the application by using a JAAS configuration file and using Spring Boot properties.

**Using JAAS Configuration Files**

The JAAS and (optionally) krb5 file locations can be set for Spring Cloud Stream applications by using system properties. The following example shows how to launch a Spring Cloud Stream application with SASL and Kerberos by using a JAAS configuration file:

java -Djava.security.auth.login.config=/path.to/kafka\_client\_jaas.conf -jar log.jar \

--spring.cloud.stream.kafka.binder.brokers=secure.server:9092 \

--spring.cloud.stream.bindings.input.destination=stream.ticktock \

--spring.cloud.stream.kafka.binder.configuration.security.protocol=SASL\_PLAINTEXT

**Using Spring Boot Properties**

As an alternative to having a JAAS configuration file, Spring Cloud Stream provides a mechanism for setting up the JAAS configuration for Spring Cloud Stream applications by using Spring Boot properties.

The following properties can be used to configure the login context of the Kafka client:

spring.cloud.stream.kafka.binder.jaas.loginModule

The login module name. Not necessary to be set in normal cases.

Default: com.sun.security.auth.module.Krb5LoginModule.

spring.cloud.stream.kafka.binder.jaas.controlFlag

The control flag of the login module.

Default: required.

spring.cloud.stream.kafka.binder.jaas.options

Map with a key/value pair containing the login module options.

Default: Empty map.

The following example shows how to launch a Spring Cloud Stream application with SASL and Kerberos by using Spring Boot configuration properties:

java --spring.cloud.stream.kafka.binder.brokers=secure.server:9092 \

--spring.cloud.stream.bindings.input.destination=stream.ticktock \

--spring.cloud.stream.kafka.binder.autoCreateTopics=false \

--spring.cloud.stream.kafka.binder.configuration.security.protocol=SASL\_PLAINTEXT \

--spring.cloud.stream.kafka.binder.jaas.options.useKeyTab=true \

--spring.cloud.stream.kafka.binder.jaas.options.storeKey=true \

--spring.cloud.stream.kafka.binder.jaas.options.keyTab=/etc/security/keytabs/kafka\_client.keytab \

--spring.cloud.stream.kafka.binder.jaas.options.principal=kafka-client-1@EXAMPLE.COM

The preceding example represents the equivalent of the following JAAS file:

KafkaClient {

com.sun.security.auth.module.Krb5LoginModule required

useKeyTab=true

storeKey=true

keyTab="/etc/security/keytabs/kafka\_client.keytab"

principal="kafka-client-1@EXAMPLE.COM";

};

If the topics required already exist on the broker or will be created by an administrator, autocreation can be turned off and only client JAAS properties need to be sent.

|  |
| --- |
| [Note] |
| Do not mix JAAS configuration files and Spring Boot properties in the same application. If the -Djava.security.auth.login.config system property is already present, Spring Cloud Stream ignores the Spring Boot properties. |
| [Note] |
| Be careful when using the autoCreateTopics and autoAddPartitions with Kerberos. Usually, applications may use principals that do not have administrative rights in Kafka and Zookeeper. Consequently, relying on Spring Cloud Stream to create/modify topics may fail. In secure environments, we strongly recommend creating topics and managing ACLs administratively by using Kafka tooling. | |

**Example: Pausing and Resuming the Consumer**

If you wish to suspend consumption but not cause a partition rebalance, you can pause and resume the consumer. This is facilitated by adding the Consumer as a parameter to your @StreamListener. To resume, you need an ApplicationListener for ListenerContainerIdleEvent instances. The frequency at which events are published is controlled by the idleEventInterval property. Since the consumer is not thread-safe, you must call these methods on the calling thread.

The following simple application shows how to pause and resume:

*@SpringBootApplication*

*@EnableBinding(Sink.class)*

**public** **class** Application {

**public** **static** **void** main(String[] args) {

SpringApplication.run(Application.**class**, args);

}

*@StreamListener(Sink.INPUT)*

**public** **void** in(String in, *@Header(KafkaHeaders.CONSUMER)* Consumer<?, ?> consumer) {

System.out.println(in);

consumer.pause(Collections.singleton(**new** TopicPartition("myTopic", 0)));

}

*@Bean*

**public** ApplicationListener<ListenerContainerIdleEvent> idleListener() {

**return** event -> {

System.out.println(event);

**if** (event.getConsumer().paused().size() > 0) {

event.getConsumer().resume(event.getConsumer().paused());

}

};

}

}

**37.4 Error Channels**

Starting with version 1.3, the binder unconditionally sends exceptions to an error channel for each consumer destination and can also be configured to send async producer send failures to an error channel. See [Section 27.4, “Error Handling”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-overview-error-handling) for more information.

The payload of the ErrorMessage for a send failure is a KafkaSendFailureException with properties:

* failedMessage: The Spring Messaging Message<?> that failed to be sent.
* record: The raw ProducerRecord that was created from the failedMessage

There is no automatic handling of producer exceptions (such as sending to a [Dead-Letter queue](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#kafka-dlq-processing)). You can consume these exceptions with your own Spring Integration flow.

**37.5 Kafka Metrics**

Kafka binder module exposes the following metrics:

spring.cloud.stream.binder.kafka.someGroup.someTopic.lag: This metric indicates how many messages have not been yet consumed from a given binder’s topic by a given consumer group. For example, if the value of the metric spring.cloud.stream.binder.kafka.myGroup.myTopic.lag is 1000, the consumer group named myGroup has 1000 messages waiting to be consumed from the topic calle myTopic. This metric is particularly useful for providing auto-scaling feedback to a PaaS platform.

**37.6 Dead-Letter Topic Processing**

Because you cannot anticipate how users would want to dispose of dead-lettered messages, the framework does not provide any standard mechanism to handle them. If the reason for the dead-lettering is transient, you may wish to route the messages back to the original topic. However, if the problem is a permanent issue, that could cause an infinite loop. The sample Spring Boot application within this topic is an example of how to route those messages back to the original topic, but it moves them to a “parking lot” topic after three attempts. The application is another spring-cloud-stream application that reads from the dead-letter topic. It terminates when no messages are received for 5 seconds.

The examples assume the original destination is so8400out and the consumer group is so8400.

There are a couple of strategies to consider:

* Consider running the rerouting only when the main application is not running. Otherwise, the retries for transient errors are used up very quickly.
* Alternatively, use a two-stage approach: Use this application to route to a third topic and another to route from there back to the main topic.

The following code listings show the sample application:

**application.properties.**

spring.cloud.stream.bindings.input.group=so8400replay

spring.cloud.stream.bindings.input.destination=error.so8400out.so8400

spring.cloud.stream.bindings.output.destination=so8400out

spring.cloud.stream.bindings.output.producer.partitioned=true

spring.cloud.stream.bindings.parkingLot.destination=so8400in.parkingLot

spring.cloud.stream.bindings.parkingLot.producer.partitioned=true

spring.cloud.stream.kafka.binder.configuration.auto.offset.reset=earliest

spring.cloud.stream.kafka.binder.headers=x-retries

**Application.**

*@SpringBootApplication*

*@EnableBinding(TwoOutputProcessor.class)*

**public** **class** ReRouteDlqKApplication **implements** CommandLineRunner {

**private** **static** **final** String X\_RETRIES\_HEADER = "x-retries";

**public** **static** **void** main(String[] args) {

SpringApplication.run(ReRouteDlqKApplication.**class**, args).close();

}

**private** **final** AtomicInteger processed = **new** AtomicInteger();

*@Autowired*

**private** MessageChannel parkingLot;

*@StreamListener(Processor.INPUT)*

*@SendTo(Processor.OUTPUT)*

**public** Message<?> reRoute(Message<?> failed) {

processed.incrementAndGet();

Integer retries = failed.getHeaders().get(X\_RETRIES\_HEADER, Integer.**class**);

**if** (retries == null) {

System.out.println("First retry for " + failed);

**return** MessageBuilder.fromMessage(failed)

.setHeader(X\_RETRIES\_HEADER, **new** Integer(1))

.setHeader(BinderHeaders.PARTITION\_OVERRIDE,

failed.getHeaders().get(KafkaHeaders.RECEIVED\_PARTITION\_ID))

.build();

}

**else** **if** (retries.intValue() < 3) {

System.out.println("Another retry for " + failed);

**return** MessageBuilder.fromMessage(failed)

.setHeader(X\_RETRIES\_HEADER, **new** Integer(retries.intValue() + 1))

.setHeader(BinderHeaders.PARTITION\_OVERRIDE,

failed.getHeaders().get(KafkaHeaders.RECEIVED\_PARTITION\_ID))

.build();

}

**else** {

System.out.println("Retries exhausted for " + failed);

parkingLot.send(MessageBuilder.fromMessage(failed)

.setHeader(BinderHeaders.PARTITION\_OVERRIDE,

failed.getHeaders().get(KafkaHeaders.RECEIVED\_PARTITION\_ID))

.build());

}

**return** null;

}

*@Override*

**public** **void** run(String... args) **throws** Exception {

**while** (true) {

**int** count = **this**.processed.get();

Thread.sleep(5000);

**if** (count == **this**.processed.get()) {

System.out.println("Idle, terminating");

**return**;

}

}

}

**public** **interface** TwoOutputProcessor **extends** Processor {

*@Output("parkingLot")*

MessageChannel parkingLot();

}

}

**37.7 Partitioning with the Kafka Binder**

Apache Kafka supports topic partitioning natively.

Sometimes it is advantageous to send data to specific partitions — for example, when you want to strictly order message processing (all messages for a particular customer should go to the same partition).

The following example shows how to configure the producer and consumer side:

*@SpringBootApplication*

*@EnableBinding(Source.class)*

**public** **class** KafkaPartitionProducerApplication {

**private** **static** **final** Random RANDOM = **new** Random(System.currentTimeMillis());

**private** **static** **final** String[] data = **new** String[] {

"foo1", "bar1", "qux1",

"foo2", "bar2", "qux2",

"foo3", "bar3", "qux3",

"foo4", "bar4", "qux4",

};

**public** **static** **void** main(String[] args) {

**new** SpringApplicationBuilder(KafkaPartitionProducerApplication.**class**)

.web(false)

.run(args);

}

*@InboundChannelAdapter(channel = Source.OUTPUT, poller = @Poller(fixedRate = "5000"))*

**public** Message<?> generate() {

String value = data[RANDOM.nextInt(data.length)];

System.out.println("Sending: " + value);

**return** MessageBuilder.withPayload(value)

.setHeader("partitionKey", value)

.build();

}

}

**application.yml.**

spring:

cloud:

stream:

bindings:

output:

destination: partitioned.topic

producer:

partitioned: **true**

partition-key-expression: headers['partitionKey'**]**

partition-count: 12

|  |  |  |
| --- | --- | --- |
| [Important] | | **Important** |
| The topic must be provisioned to have enough partitions to achieve the desired concurrency for all consumer groups. The above configuration supports up to 12 consumer instances (6 if their concurrency is 2, 4 if their concurrency is 3, and so on). It is generally best to “over-provision” the partitions to allow for future increases in consumers or concurrency. |
| [Note] |
| The preceding configuration uses the default partitioning (key.hashCode() % partitionCount). This may or may not provide a suitably balanced algorithm, depending on the key values. You can override this default by using the partitionSelectorExpression or partitionSelectorClassproperties. | | |

Since partitions are natively handled by Kafka, no special configuration is needed on the consumer side. Kafka allocates partitions across the instances.

The following Spring Boot application listens to a Kafka stream and prints (to the console) the partition ID to which each message goes:

*@SpringBootApplication*

*@EnableBinding(Sink.class)*

**public** **class** KafkaPartitionConsumerApplication {

**public** **static** **void** main(String[] args) {

**new** SpringApplicationBuilder(KafkaPartitionConsumerApplication.**class**)

.web(false)

.run(args);

}

*@StreamListener(Sink.INPUT)*

**public** **void** listen(*@Payload* String in, *@Header(KafkaHeaders.RECEIVED\_PARTITION\_ID)* **int** partition) {

System.out.println(in + " received from partition " + partition);

}

}

**application.yml.**

spring:

cloud:

stream:

bindings:

input:

destination: partitioned.topic

group: myGroup

You can add instances as needed. Kafka rebalances the partition allocations. If the instance count (or instance count \* concurrency) exceeds the number of partitions, some consumers are idle.

**38. Apache Kafka Streams Binder**

**38.1 Usage**

For using the Kafka Streams binder, you just need to add it to your Spring Cloud Stream application, using the following Maven coordinates:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-stream-binder-kafka-streams</artifactId>

</dependency>

**38.2 Kafka Streams Binder Overview**

Spring Cloud Stream’s Apache Kafka support also includes a binder implementation designed explicitly for Apache Kafka Streams binding. With this native integration, a Spring Cloud Stream "processor" application can directly use the [Apache Kafka Streams](https://kafka.apache.org/documentation/streams/developer-guide) APIs in the core business logic.

Kafka Streams binder implementation builds on the foundation provided by the [Kafka Streams in Spring Kafka](https://docs.spring.io/spring-kafka/reference/html/_reference.html#kafka-streams) project.

As part of this native integration, the high-level [Streams DSL](https://docs.confluent.io/current/streams/developer-guide/dsl-api.html) provided by the Kafka Streams API is available for use in the business logic, too.

An early version of the [Processor API](https://docs.confluent.io/current/streams/developer-guide/processor-api.html) support is available as well.

As noted early-on, Kafka Streams support in Spring Cloud Stream strictly only available for use in the Processor model. A model in which the messages read from an inbound topic, business processing can be applied, and the transformed messages can be written to an outbound topic. It can also be used in Processor applications with a no-outbound destination.

**38.2.1 Streams DSL**

This application consumes data from a Kafka topic (e.g., words), computes word count for each unique word in a 5 seconds time window, and the computed results are sent to a downstream topic (e.g., counts) for further processing.

@SpringBootApplication

@EnableBinding(KStreamProcessor.class)

public class WordCountProcessorApplication {

@StreamListener("input")

@SendTo("output")

public KStream<?, WordCount> process(KStream<?, String> input) {

return input

.flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))

.groupBy((key, value) -> value)

.windowedBy(TimeWindows.of(5000))

.count(Materialized.as("WordCounts-multi"))

.toStream()

.map((key, value) -> new KeyValue<>(null, new WordCount(key.key(), value, new Date(key.window().start()), new Date(key.window().end()))));

}

public static void main(String[] args) {

SpringApplication.run(WordCountProcessorApplication.class, args);

}

Once built as a uber-jar (e.g., wordcount-processor.jar), you can run the above example like the following.

java -jar wordcount-processor.jar --spring.cloud.stream.bindings.input.destination=words --spring.cloud.stream.bindings.output.destination=counts

This application will consume messages from the Kafka topic words and the computed results are published to an output topic counts.

Spring Cloud Stream will ensure that the messages from both the incoming and outgoing topics are automatically bound as KStream objects. As a developer, you can exclusively focus on the business aspects of the code, i.e. writing the logic required in the processor. Setting up the Streams DSL specific configuration required by the Kafka Streams infrastructure is automatically handled by the framework.

**38.3 Configuration Options**

This section contains the configuration options used by the Kafka Streams binder.

For common configuration options and properties pertaining to binder, refer to the [core documentation](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#binding-properties).

**38.3.1 Kafka Streams Properties**

The following properties are available at the binder level and must be prefixed with spring.cloud.stream.kafka.streams.binder. literal.

configuration

Map with a key/value pair containing properties pertaining to Apache Kafka Streams API. This property must be prefixed with spring.cloud.stream.kafka.streams.binder.. Following are some examples of using this property.

spring.cloud.stream.kafka.streams.binder.configuration.default.key.serde=org.apache.kafka.common.serialization.Serdes$StringSerde

spring.cloud.stream.kafka.streams.binder.configuration.default.value.serde=org.apache.kafka.common.serialization.Serdes$StringSerde

spring.cloud.stream.kafka.streams.binder.configuration.commit.interval.ms=1000

For more information about all the properties that may go into streams configuration, see StreamsConfig JavaDocs in Apache Kafka Streams docs.

brokers

Broker URL

Default: localhost

zkNodes

Zookeeper URL

Default: localhost

serdeError

Deserialization error handler type. Possible values are - logAndContinue, logAndFail or sendToDlq

Default: logAndFail

applicationId

Application ID for all the stream configurations in the current application context. You can override the application id for an individual StreamListener method using the group property on the binding. You have to ensure that you are using the same group name for all input bindings in the case of multiple inputs on the same methods.

Default: default

The following properties are *only* available for Kafka Streams producers and must be prefixed with spring.cloud.stream.kafka.streams.bindings.<binding name>.producer. literal.

keySerde

key serde to use

Default: none.

valueSerde

value serde to use

Default: none.

useNativeEncoding

flag to enable native encoding

Default: false.

The following properties are *only* available for Kafka Streams consumers and must be prefixed with spring.cloud.stream.kafka.streams.bindings.<binding name>.consumer. literal.

keySerde

key serde to use

Default: none.

valueSerde

value serde to use

Default: none.

materializedAs

state store to materialize when using incoming KTable types

Default: none.

useNativeDecoding

flag to enable native decoding

Default: false.

dlqName

DLQ topic name.

Default: none.

**38.3.2 TimeWindow properties:**

Windowing is an important concept in stream processing applications. Following properties are available to configure time-window computations.

spring.cloud.stream.kafka.streams.timeWindow.length

When this property is given, you can autowire a TimeWindows bean into the application. The value is expressed in milliseconds.

Default: none.

spring.cloud.stream.kafka.streams.timeWindow.advanceBy

Value is given in milliseconds.

Default: none.

**38.4 Multiple Input Bindings**

For use cases that requires multiple incoming KStream objects or a combination of KStream and KTable objects, the Kafka Streams binder provides multiple bindings support.

Let’s see it in action.

**38.4.1 Multiple Input Bindings as a Sink**

@EnableBinding(KStreamKTableBinding.class)

.....

.....

@StreamListener

public void process(@Input("inputStream") KStream<String, PlayEvent> playEvents,

@Input("inputTable") KTable<Long, Song> songTable) {

....

....

}

interface KStreamKTableBinding {

@Input("inputStream")

KStream<?, ?> inputStream();

@Input("inputTable")

KTable<?, ?> inputTable();

}

In the above example, the application is written as a sink, i.e. there are no output bindings and the application has to decide concerning downstream processing. When you write applications in this style, you might want to send the information downstream or store them in a state store (See below for Queryable State Stores).

In the case of incoming KTable, if you want to materialize the computations to a state store, you have to express it through the following property.

spring.cloud.stream.kafka.streams.bindings.inputTable.consumer.materializedAs: all-songs

**38.4.2 Multiple Input Bindings as a Processor**

@EnableBinding(KStreamKTableBinding.class)

....

....

@StreamListener

@SendTo("output")

public KStream<String, Long> process(@Input("input") KStream<String, Long> userClicksStream,

@Input("inputTable") KTable<String, String> userRegionsTable) {

....

....

}

interface KStreamKTableBinding extends KafkaStreamsProcessor {

@Input("inputX")

KTable<?, ?> inputTable();

}

**38.5 Multiple Output Bindings (aka Branching)**

Kafka Streams allow outbound data to be split into multiple topics based on some predicates. The Kafka Streams binder provides support for this feature without compromising the programming model exposed through StreamListener in the end user application.

You can write the application in the usual way as demonstrated above in the word count example. However, when using the branching feature, you are required to do a few things. First, you need to make sure that your return type is KStream[] instead of a regular KStream. Second, you need to use the SendTo annotation containing the output bindings in the order (see example below). For each of these output bindings, you need to configure destination, content-type etc., complying with the standard Spring Cloud Stream expectations.

Here is an example:

@EnableBinding(KStreamProcessorWithBranches.class)

@EnableAutoConfiguration

public static class WordCountProcessorApplication {

@Autowired

private TimeWindows timeWindows;

@StreamListener("input")

@SendTo({"output1","output2","output3})

public KStream<?, WordCount>[] process(KStream<Object, String> input) {

Predicate<Object, WordCount> isEnglish = (k, v) -> v.word.equals("english");

Predicate<Object, WordCount> isFrench = (k, v) -> v.word.equals("french");

Predicate<Object, WordCount> isSpanish = (k, v) -> v.word.equals("spanish");

return input

.flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))

.groupBy((key, value) -> value)

.windowedBy(timeWindows)

.count(Materialized.as("WordCounts-1"))

.toStream()

.map((key, value) -> new KeyValue<>(null, new WordCount(key.key(), value, new Date(key.window().start()), new Date(key.window().end()))))

.branch(isEnglish, isFrench, isSpanish);

}

interface KStreamProcessorWithBranches {

@Input("input")

KStream<?, ?> input();

@Output("output1")

KStream<?, ?> output1();

@Output("output2")

KStream<?, ?> output2();

@Output("output3")

KStream<?, ?> output3();

}

}

Properties:

spring.cloud.stream.bindings.output1.contentType: application/json

spring.cloud.stream.bindings.output2.contentType: application/json

spring.cloud.stream.bindings.output3.contentType: application/json

spring.cloud.stream.kafka.streams.binder.configuration.commit.interval.ms: 1000

spring.cloud.stream.kafka.streams.binder.configuration:

default.key.serde: org.apache.kafka.common.serialization.Serdes$StringSerde

default.value.serde: org.apache.kafka.common.serialization.Serdes$StringSerde

spring.cloud.stream.bindings.output1:

destination: foo

producer:

headerMode: raw

spring.cloud.stream.bindings.output2:

destination: bar

producer:

headerMode: raw

spring.cloud.stream.bindings.output3:

destination: fox

producer:

headerMode: raw

spring.cloud.stream.bindings.input:

destination: words

consumer:

headerMode: raw

**38.6 Message Conversion**

Similar to message-channel based binder applications, the Kafka Streams binder adapts to the out-of-the-box content-type conversions without any compromise.

It is typical for Kafka Streams operations to know the type of SerDe’s used to transform the key and value correctly. Therefore, it may be more natural to rely on the SerDe facilities provided by the Apache Kafka Streams library itself at the inbound and outbound conversions rather than using the content-type conversions offered by the framework. On the other hand, you might be already familiar with the content-type conversion patterns provided by the framework, and that, you’d like to continue using for inbound and outbound conversions.

Both the options are supported in the Kafka Streams binder implementation.

**38.6.1 Outbound serialization**

If native encoding is disabled (which is the default), then the framework will convert the message using the contentType set by the user (otherwise, the default application/json will be applied). It will ignore any SerDe set on the outbound in this case for outbound serialization.

Here is the property to set the contentType on the outbound.

spring.cloud.stream.bindings.output.contentType: application/json

Here is the property to enable native encoding.

spring.cloud.stream.bindings.output.nativeEncoding: true

If native encoding is enabled on the output binding (user has to enable it as above explicitly), then the framework will skip any form of automatic message conversion on the outbound. In that case, it will switch to the Serde set by the user. The valueSerde property set on the actual output binding will be used. Here is an example.

spring.cloud.stream.kafka.streams.bindings.output.producer.valueSerde: org.apache.kafka.common.serialization.Serdes$StringSerde

If this property is not set, then it will use the "default" SerDe: spring.cloud.stream.kafka.streams.binder.configuration.default.value.serde.

It is worth to mention that Kafka Streams binder does not serialize the keys on outbound - it simply relies on Kafka itself. Therefore, you either have to specify the keySerde property on the binding or it will default to the application-wide common keySerde.

Binding level key serde:

spring.cloud.stream.kafka.streams.bindings.output.producer.keySerde

Common Key serde:

spring.cloud.stream.kafka.streams.binder.configuration.default.key.serde

If branching is used, then you need to use multiple output bindings. For example,

interface KStreamProcessorWithBranches {

@Input("input")

KStream<?, ?> input();

@Output("output1")

KStream<?, ?> output1();

@Output("output2")

KStream<?, ?> output2();

@Output("output3")

KStream<?, ?> output3();

}

If nativeEncoding is set, then you can set different SerDe’s on individual output bindings as below.

spring.cloud.stream.kafka.streams.bindings.output1.producer.valueSerde=IntegerSerde

spring.cloud.stream.kafka.streams.bindings.output2.producer.valueSerde=StringSerde

spring.cloud.stream.kafka.streams.bindings.output3.producer.valueSerde=JsonSerde

Then if you have SendTo like this, @SendTo({"output1", "output2", "output3"}), the KStream[] from the branches are applied with proper SerDe objects as defined above. If you are not enabling nativeEncoding, you can then set different contentType values on the output bindings as below. In that case, the framework will use the appropriate message converter to convert the messages before sending to Kafka.

spring.cloud.stream.bindings.output1.contentType: application/json

spring.cloud.stream.bindings.output2.contentType: application/java-serialzied-object

spring.cloud.stream.bindings.output3.contentType: application/octet-stream

**38.6.2 Inbound Deserialization**

Similar rules apply to data deserialization on the inbound.

If native decoding is disabled (which is the default), then the framework will convert the message using the contentType set by the user (otherwise, the default application/json will be applied). It will ignore any SerDe set on the inbound in this case for inbound deserialization.

Here is the property to set the contentType on the inbound.

spring.cloud.stream.bindings.input.contentType: application/json

Here is the property to enable native decoding.

spring.cloud.stream.bindings.input.nativeDecoding: true

If native decoding is enabled on the input binding (user has to enable it as above explicitly), then the framework will skip doing any message conversion on the inbound. In that case, it will switch to the SerDe set by the user. The valueSerde property set on the actual output binding will be used. Here is an example.

spring.cloud.stream.kafka.streams.bindings.input.consumer.valueSerde: org.apache.kafka.common.serialization.Serdes$StringSerde

If this property is not set, it will use the default SerDe: spring.cloud.stream.kafka.streams.binder.configuration.default.value.serde.

It is worth to mention that Kafka Streams binder does not deserialize the keys on inbound - it simply relies on Kafka itself. Therefore, you either have to specify the keySerde property on the binding or it will default to the application-wide common keySerde.

Binding level key serde:

spring.cloud.stream.kafka.streams.bindings.input.consumer.keySerde

Common Key serde:

spring.cloud.stream.kafka.streams.binder.configuration.default.key.serde

As in the case of KStream branching on the outbound, the benefit of setting value SerDe per binding is that if you have multiple input bindings (multiple KStreams object) and they all require separate value SerDe’s, then you can configure them individually. If you use the common configuration approach, then this feature won’t be applicable.

**38.7 Error Handling**

Apache Kafka Streams provide the capability for natively handling exceptions from deserialization errors. For details on this support, please see [this](https://cwiki.apache.org/confluence/display/KAFKA/KIP-161%3A+streams+deserialization+exception+handlers) Out of the box, Apache Kafka Streams provide two kinds of deserialization exception handlers - logAndContinue and logAndFail. As the name indicates, the former will log the error and continue processing the next records and the latter will log the error and fail. LogAndFail is the default deserialization exception handler.

**38.7.1 Handling Deserialization Exceptions**

Kafka Streams binder supports a selection of exception handlers through the following properties.

spring.cloud.stream.kafka.streams.binder.serdeError: logAndContinue

In addition to the above two deserialization exception handlers, the binder also provides a third one for sending the erroneous records (poison pills) to a DLQ topic. Here is how you enable this DLQ exception handler.

spring.cloud.stream.kafka.streams.binder.serdeError: sendToDlq

When the above property is set, all the deserialization error records are automatically sent to the DLQ topic.

spring.cloud.stream.kafka.streams.bindings.input.consumer.dlqName: foo-dlq

If this is set, then the error records are sent to the topic foo-dlq. If this is not set, then it will create a DLQ topic with the name error.<input-topic-name>.<group-name>.

A couple of things to keep in mind when using the exception handling feature in Kafka Streams binder.

* The property spring.cloud.stream.kafka.streams.binder.serdeError is applicable for the entire application. This implies that if there are multiple StreamListener methods in the same application, this property is applied to all of them.
* The exception handling for deserialization works consistently with native deserialization and framework provided message conversion.

**38.7.2 Handling Non-Deserialization Exceptions**

For general error handling in Kafka Streams binder, it is up to the end user applications to handle application level errors. As a side effect of providing a DLQ for deserialization exception handlers, Kafka Streams binder provides a way to get access to the DLQ sending bean directly from your application. Once you get access to that bean, you can programmatically send any exception records from your application to the DLQ.

It continues to remain hard to robust error handling using the high-level DSL; Kafka Streams doesn’t natively support error handling yet.

However, when you use the low-level Processor API in your application, there are options to control this behavior. See below.

@Autowired

private SendToDlqAndContinue dlqHandler;

@StreamListener("input")

@SendTo("output")

public KStream<?, WordCount> process(KStream<Object, String> input) {

input.process(() -> new Processor() {

ProcessorContext context;

@Override

public void init(ProcessorContext context) {

this.context = context;

}

@Override

public void process(Object o, Object o2) {

try {

.....

.....

}

catch(Exception e) {

//explicitly provide the kafka topic corresponding to the input binding as the first argument.

//DLQ handler will correctly map to the dlq topic from the actual incoming destination.

dlqHandler.sendToDlq("topic-name", (byte[]) o1, (byte[]) o2, context.partition());

}

}

.....

.....

});

}

**38.8 Interactive Queries**

As part of the public Kafka Streams binder API, we expose a class called QueryableStoreRegistry. You can access this as a Spring bean in your application. An easy way to get access to this bean from your application is to "autowire" the bean in your application.

@Autowired

private QueryableStoreRegistry queryableStoreRegistry;

Once you gain access to this bean, then you can query for the particular state-store that you are interested. See below.

ReadOnlyKeyValueStore<Object, Object> keyValueStore =

queryableStoreRegistry.getQueryableStoreType("my-store", QueryableStoreTypes.keyValueStore());

**38.9 Accessing the underlying KafkaStreams object**

StreamBuilderFactoryBean from spring-kafka that is responsible for constructing the KafkaStreams object can be accessed programmatically. Each StreamBuilderFactoryBean is registered as stream-builder and appended with the StreamListener method name. If your StreamListener method is named as process for example, the stream builder bean is named as stream-builder-process. Since this is a factory bean, it should be accessed by prepending an ampersand (&) when accessing it programmatically. Following is an example and it assumes the StreamListener method is named as process

StreamsBuilderFactoryBean streamsBuilderFactoryBean = context.getBean("&stream-builder-process", StreamsBuilderFactoryBean.class);

KafkaStreams kafkaStreams = streamsBuilderFactoryBean.getKafkaStreams();

**39. RabbitMQ Binder**

**39.1 Usage**

To use the RabbitMQ binder, you can add it to your Spring Cloud Stream application, by using the following Maven coordinates:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-stream-binder-rabbit</artifactId>

</dependency>

Alternatively, you can use the Spring Cloud Stream RabbitMQ Starter, as follows:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-stream-rabbit</artifactId>

</dependency>

**39.2 RabbitMQ Binder Overview**

The following simplified diagram shows how the RabbitMQ binder operates:

**Figure 39.1. RabbitMQ Binder**

By default, the RabbitMQ Binder implementation maps each destination to a TopicExchange. For each consumer group, a Queue is bound to that TopicExchange. Each consumer instance has a corresponding RabbitMQ Consumer instance for its group’s Queue. For partitioned producers and consumers, the queues are suffixed with the partition index and use the partition index as the routing key. For anonymous consumers (those with no group property), an auto-delete queue (with a randomized unique name) is used.

By using the optional autoBindDlq option, you can configure the binder to create and configure dead-letter queues (DLQs) (and a dead-letter exchange DLX, as well as routing infrastructure). By default, the dead letter queue has the name of the destination, appended with .dlq. If retry is enabled (maxAttempts > 1), failed messages are delivered to the DLQ after retries are exhausted. If retry is disabled (maxAttempts = 1), you should set requeueRejected to false (the default) so that failed messages are routed to the DLQ, instead of being re-queued. In addition, republishToDlq causes the binder to publish a failed message to the DLQ (instead of rejecting it). This feature lets additional information (such as the stack trace in the x-exception-stacktrace header) be added to the message in headers. This option does not need retry enabled. You can republish a failed message after just one attempt. Starting with version 1.2, you can configure the delivery mode of republished messages. See the [republishDeliveryMode property](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-stream-rabbit-republish-delivery-mode).

|  |  |
| --- | --- |
| [Important] | **Important** |
| Setting requeueRejected to true (with republishToDlq=false ) causes the message to be re-queued and redelivered continually, which is likely not what you want unless the reason for the failure is transient. In general, you should enable retry within the binder by setting maxAttempts to greater than one or by setting republishToDlq to true. |

See [Section 39.3.1, “RabbitMQ Binder Properties”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#rabbit-binder-properties) for more information about these properties.

The framework does not provide any standard mechanism to consume dead-letter messages (or to re-route them back to the primary queue). Some options are described in [Section 39.6, “Dead-Letter Queue Processing”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#rabbit-dlq-processing).

|  |
| --- |
| [Note] |
| When multiple RabbitMQ binders are used in a Spring Cloud Stream application, it is important to disable 'RabbitAutoConfiguration' to avoid the same configuration from RabbitAutoConfiguration being applied to the two binders. You can exclude the class by using the @SpringBootApplicationannotation. |

Starting with version 2.0, the RabbitMessageChannelBinder sets the RabbitTemplate.userPublisherConnection property to true so that the non-transactional producers avoid deadlocks on consumers, which can happen if cached connections are blocked because of a [memory alarm](https://www.rabbitmq.com/memory.html) on the broker.

**39.3 Configuration Options**

This section contains settings specific to the RabbitMQ Binder and bound channels.

For general binding configuration options and properties, see the [Spring Cloud Stream core documentation](https://github.com/spring-cloud/spring-cloud-stream/blob/master/spring-cloud-stream-core-docs/src/main/asciidoc/spring-cloud-stream-overview.adoc#configuration-options).

**39.3.1 RabbitMQ Binder Properties**

By default, the RabbitMQ binder uses Spring Boot’s ConnectionFactory. Conseuqently, it supports all Spring Boot configuration options for RabbitMQ. (For reference, see the [Spring Boot documentation](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#common-application-properties)). RabbitMQ configuration options use the spring.rabbitmq prefix.

In addition to Spring Boot options, the RabbitMQ binder supports the following properties:

spring.cloud.stream.rabbit.binder.adminAddresses

A comma-separated list of RabbitMQ management plugin URLs. Only used when nodes contains more than one entry. Each entry in this list must have a corresponding entry in spring.rabbitmq.addresses. Only needed if you use a RabbitMQ cluster and wish to consume from the node that hosts the queue. See [Queue Affinity and the LocalizedQueueConnectionFactory](https://docs.spring.io/spring-amqp/reference/html/_reference.html#queue-affinity) for more information.

Default: empty.

spring.cloud.stream.rabbit.binder.nodes

A comma-separated list of RabbitMQ node names. When more than one entry, used to locate the server address where a queue is located. Each entry in this list must have a corresponding entry in spring.rabbitmq.addresses. Only needed if you use a RabbitMQ cluster and wish to consume from the node that hosts the queue. See [Queue Affinity and the LocalizedQueueConnectionFactory](https://docs.spring.io/spring-amqp/reference/html/_reference.html#queue-affinity) for more information.

Default: empty.

spring.cloud.stream.rabbit.binder.compressionLevel

The compression level for compressed bindings. See java.util.zip.Deflater.

Default: 1 (BEST\_LEVEL).

spring.cloud.stream.binder.connection-name-prefix

A connection name prefix used to name the connection(s) created by this binder. The name is this prefix followed by #n, where n increments each time a new connection is opened.

Default: none (Spring AMQP default).

**39.3.2 RabbitMQ Consumer Properties**

The following properties are available for Rabbit consumers only and must be prefixed with spring.cloud.stream.rabbit.bindings.<channelName>.consumer..

acknowledgeMode

The acknowledge mode.

Default: AUTO.

autoBindDlq

Whether to automatically declare the DLQ and bind it to the binder DLX.

Default: false.

bindingRoutingKey

The routing key with which to bind the queue to the exchange (if bindQueue is true). For partitioned destinations, -<instanceIndex> is appended.

Default: #.

bindQueue

Whether to bind the queue to the destination exchange. Set it to false if you have set up your own infrastructure and have previously created and bound the queue.

Default: true.

deadLetterQueueName

The name of the DLQ

Default: prefix+destination.dlq

deadLetterExchange

A DLX to assign to the queue. Relevant only if autoBindDlq is true.

Default: 'prefix+DLX'

deadLetterRoutingKey

A dead letter routing key to assign to the queue. Relevant only if autoBindDlq is true.

Default: destination

declareExchange

Whether to declare the exchange for the destination.

Default: true.

delayedExchange

Whether to declare the exchange as a Delayed Message Exchange. Requires the delayed message exchange plugin on the broker. The x-delayed-typeargument is set to the exchangeType.

Default: false.

dlqDeadLetterExchange

If a DLQ is declared, a DLX to assign to that queue.

Default: none

dlqDeadLetterRoutingKey

If a DLQ is declared, a dead letter routing key to assign to that queue.

Default: none

dlqExpires

How long before an unused dead letter queue is deleted (in milliseconds).

Default: no expiration

dlqLazy

Declare the dead letter queue with the x-queue-mode=lazy argument. See [“Lazy Queues”](https://www.rabbitmq.com/lazy-queues.html). Consider using a policy instead of this setting, because using a policy allows changing the setting without deleting the queue.

Default: false.

dlqMaxLength

Maximum number of messages in the dead letter queue.

Default: no limit

dlqMaxLengthBytes

Maximum number of total bytes in the dead letter queue from all messages.

Default: no limit

dlqMaxPriority

Maximum priority of messages in the dead letter queue (0-255).

Default: none

dlqTtl

Default time to live to apply to the dead letter queue when declared (in milliseconds).

Default: no limit

durableSubscription

Whether the subscription should be durable. Only effective if group is also set.

Default: true.

exchangeAutoDelete

If declareExchange is true, whether the exchange should be auto-deleted (that is, removed after the last queue is removed).

Default: true.

exchangeDurable

If declareExchange is true, whether the exchange should be durable (that is, it survives broker restart).

Default: true.

exchangeType

The exchange type: direct, fanout or topic for non-partitioned destinations and direct or topic for partitioned destinations.

Default: topic.

exclusive

Whether to create an exclusive consumer. Concurrency should be 1 when this is true. Often used when strict ordering is required but enabling a hot standby instance to take over after a failure. See recoveryInterval, which controls how often a standby instance attempts to consume.

Default: false.

expires

How long before an unused queue is deleted (in milliseconds).

Default: no expiration

failedDeclarationRetryInterval

The interval (in milliseconds) between attempts to consume from a queue if it is missing.

Default: 5000

headerPatterns

Patterns for headers to be mapped from inbound messages.

Default: ['\*'] (all headers).

lazy

Declare the queue with the x-queue-mode=lazy argument. See [“Lazy Queues”](https://www.rabbitmq.com/lazy-queues.html). Consider using a policy instead of this setting, because using a policy allows changing the setting without deleting the queue.

Default: false.

maxConcurrency

The maximum number of consumers.

Default: 1.

maxLength

The maximum number of messages in the queue.

Default: no limit

maxLengthBytes

The maximum number of total bytes in the queue from all messages.

Default: no limit

maxPriority

The maximum priority of messages in the queue (0-255).

Default: none

missingQueuesFatal

When the queue cannot be found, whether to treat the condition as fatal and stop the listener container. Defaults to false so that the container keeps trying to consume from the queue — for example, when using a cluster and the node hosting a non-HA queue is down.

Default: false

prefetch

Prefetch count.

Default: 1.

prefix

A prefix to be added to the name of the destination and queues.

Default: "".

queueDeclarationRetries

The number of times to retry consuming from a queue if it is missing. Relevant only when missingQueuesFatal is true. Otherwise, the container keeps retrying indefinitely.

Default: 3

queueNameGroupOnly

When true, consume from a queue with a name equal to the group. Otherwise the queue name is destination.group. This is useful, for example, when using Spring Cloud Stream to consume from an existing RabbitMQ queue.

Default: false.

recoveryInterval

The interval between connection recovery attempts, in milliseconds.

Default: 5000.

requeueRejected

Whether delivery failures should be re-queued when retry is disabled or republishToDlq is false.

Default: false.

republishDeliveryMode

When republishToDlq is true, specifies the delivery mode of the republished message.

Default: DeliveryMode.PERSISTENT

republishToDlq

By default, messages that fail after retries are exhausted are rejected. If a dead-letter queue (DLQ) is configured, RabbitMQ routes the failed message (unchanged) to the DLQ. If set to true, the binder republishs failed messages to the DLQ with additional headers, including the exception message and stack trace from the cause of the final failure.

Default: false

transacted

Whether to use transacted channels.

Default: false.

ttl

Default time to live to apply to the queue when declared (in milliseconds).

Default: no limit

txSize

The number of deliveries between acks.

Default: 1.

**39.3.3 Rabbit Producer Properties**

The following properties are available for Rabbit producers only and must be prefixed with spring.cloud.stream.rabbit.bindings.<channelName>.producer..

autoBindDlq

Whether to automatically declare the DLQ and bind it to the binder DLX.

Default: false.

batchingEnabled

Whether to enable message batching by producers. Messages are batched into one message according to the following properties (described in the next three entries in this list): 'batchSize', batchBufferLimit, and batchTimeout. See [Batching](https://docs.spring.io/spring-amqp/reference/html/_reference.html#template-batching) for more information.

Default: false.

batchSize

The number of messages to buffer when batching is enabled.

Default: 100.

batchBufferLimit

The maximum buffer size when batching is enabled.

Default: 10000.

batchTimeout

The batch timeout when batching is enabled.

Default: 5000.

bindingRoutingKey

The routing key with which to bind the queue to the exchange (if bindQueue is true). Only applies to non-partitioned destinations. Only applies if requiredGroups are provided and then only to those groups.

Default: #.

bindQueue

Whether to bind the queue to the destination exchange. Set it to false if you have set up your own infrastructure and have previously created and bound the queue. Only applies if requiredGroups are provided and then only to those groups.

Default: true.

compress

Whether data should be compressed when sent.

Default: false.

deadLetterQueueName

The name of the DLQ Only applies if requiredGroups are provided and then only to those groups.

Default: prefix+destination.dlq

deadLetterExchange

A DLX to assign to the queue. Relevant only when autoBindDlq is true. Applies only when requiredGroups are provided and then only to those groups.

Default: 'prefix+DLX'

deadLetterRoutingKey

A dead letter routing key to assign to the queue. Relevant only when autoBindDlq is true. Applies only when requiredGroups are provided and then only to those groups.

Default: destination

declareExchange

Whether to declare the exchange for the destination.

Default: true.

delayExpression

A SpEL expression to evaluate the delay to apply to the message (x-delay header). It has no effect if the exchange is not a delayed message exchange.

Default: No x-delay header is set.

delayedExchange

Whether to declare the exchange as a Delayed Message Exchange. Requires the delayed message exchange plugin on the broker. The x-delayed-typeargument is set to the exchangeType.

Default: false.

deliveryMode

The delivery mode.

Default: PERSISTENT.

dlqDeadLetterExchange

When a DLQ is declared, a DLX to assign to that queue. Applies only if requiredGroups are provided and then only to those groups.

Default: none

dlqDeadLetterRoutingKey

When a DLQ is declared, a dead letter routing key to assign to that queue. Applies only when requiredGroups are provided and then only to those groups.

Default: none

dlqExpires

How long (in milliseconds) before an unused dead letter queue is deleted. Applies only when requiredGroups are provided and then only to those groups.

Default: no expiration

dlqLazy

Declare the dead letter queue with the x-queue-mode=lazy argument. See [“Lazy Queues”](https://www.rabbitmq.com/lazy-queues.html). Consider using a policy instead of this setting, because using a policy allows changing the setting without deleting the queue. Applies only when requiredGroups are provided and then only to those groups.

dlqMaxLength

Maximum number of messages in the dead letter queue. Applies only if requiredGroups are provided and then only to those groups.

Default: no limit

dlqMaxLengthBytes

Maximum number of total bytes in the dead letter queue from all messages. Applies only when requiredGroups are provided and then only to those groups.

Default: no limit

dlqMaxPriority

Maximum priority of messages in the dead letter queue (0-255) Applies only when requiredGroups are provided and then only to those groups.

Default: none

dlqTtl

Default time (in milliseconds) to live to apply to the dead letter queue when declared. Applies only when requiredGroups are provided and then only to those groups.

Default: no limit

exchangeAutoDelete

If declareExchange is true, whether the exchange should be auto-delete (it is removed after the last queue is removed).

Default: true.

exchangeDurable

If declareExchange is true, whether the exchange should be durable (survives broker restart).

Default: true.

exchangeType

The exchange type: direct, fanout or topic for non-partitioned destinations and direct or topic for partitioned destinations.

Default: topic.

expires

How long (in milliseconds) before an unused queue is deleted. Applies only when requiredGroups are provided and then only to those groups.

Default: no expiration

headerPatterns

Patterns for headers to be mapped to outbound messages.

Default: ['\*'] (all headers).

lazy

Declare the queue with the x-queue-mode=lazy argument. See [“Lazy Queues”](https://www.rabbitmq.com/lazy-queues.html). Consider using a policy instead of this setting, because using a policy allows changing the setting without deleting the queue. Applies only when requiredGroups are provided and then only to those groups.

Default: false.

maxLength

Maximum number of messages in the queue. Applies only when requiredGroups are provided and then only to those groups.

Default: no limit

maxLengthBytes

Maximum number of total bytes in the queue from all messages. Only applies if requiredGroups are provided and then only to those groups.

Default: no limit

maxPriority

Maximum priority of messages in the queue (0-255). Only applies if requiredGroups are provided and then only to those groups.

Default: none

prefix

A prefix to be added to the name of the destination exchange.

Default: "".

queueNameGroupOnly

When true, consume from a queue with a name equal to the group. Otherwise the queue name is destination.group. This is useful, for example, when using Spring Cloud Stream to consume from an existing RabbitMQ queue. Applies only when requiredGroups are provided and then only to those groups.

Default: false.

routingKeyExpression

A SpEL expression to determine the routing key to use when publishing messages. For a fixed routing key, use a literal expression, such as routingKeyExpression='my.routingKey' in a properties file or routingKeyExpression: '''my.routingKey''' in a YAML file.

Default: destination or destination-<partition> for partitioned destinations.

transacted

Whether to use transacted channels.

Default: false.

ttl

Default time (in milliseconds) to live to apply to the queue when declared. Applies only when requiredGroups are provided and then only to those groups.

Default: no limit

|  |
| --- |
| [Note] |
| In the case of RabbitMQ, content type headers can be set by external applications. Spring Cloud Stream supports them as part of an extended internal protocol used for any type of transport — including transports, such as Kafka (prior to 0.11), that do not natively support headers. |

**39.4 Retry With the RabbitMQ Binder**

When retry is enabled within the binder, the listener container thread is suspended for any back off periods that are configured. This might be important when strict ordering is required with a single consumer. However, for other use cases, it prevents other messages from being processed on that thread. An alternative to using binder retry is to set up dead lettering with time to live on the dead-letter queue (DLQ) as well as dead-letter configuration on the DLQ itself. See “[Section 39.3.1, “RabbitMQ Binder Properties”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#rabbit-binder-properties)” for more information about the properties discussed here. You can use the following example configuration to enable this feature:

* Set autoBindDlq to true. The binder create a DLQ. Optionally, you can specify a name in deadLetterQueueName.
* Set dlqTtl to the back off time you want to wait between redeliveries.
* Set the dlqDeadLetterExchange to the default exchange. Expired messages from the DLQ are routed to the original queue, because the default deadLetterRoutingKey is the queue name (destination.group). Setting to the default exchange is achieved by setting the property with no value, as shown in the next example.

To force a message to be dead-lettered, either throw an AmqpRejectAndDontRequeueException or set requeueRejected to true (the default) and throw any exception.

The loop continue without end, which is fine for transient problems, but you may want to give up after some number of attempts. Fortunately, RabbitMQ provides the x-death header, which lets you determine how many cycles have occurred.

To acknowledge a message after giving up, throw an ImmediateAcknowledgeAmqpException.

**39.4.1 Putting it All Together**

The following configuration creates an exchange myDestination with queue myDestination.consumerGroup bound to a topic exchange with a wildcard routing key #:

---

spring.cloud.stream.bindings.input.destination=myDestination

spring.cloud.stream.bindings.input.group=consumerGroup

#disable binder retries

spring.cloud.stream.bindings.input.consumer.max-attempts=1

#dlx/dlq setup

spring.cloud.stream.rabbit.bindings.input.consumer.auto-bind-dlq=true

spring.cloud.stream.rabbit.bindings.input.consumer.dlq-ttl=5000

spring.cloud.stream.rabbit.bindings.input.consumer.dlq-dead-letter-exchange=

---

This configuration creates a DLQ bound to a direct exchange (DLX) with a routing key of myDestination.consumerGroup. When messages are rejected, they are routed to the DLQ. After 5 seconds, the message expires and is routed to the original queue by using the queue name as the routing key, as shown in the following example:

**Spring Boot application.**

*@SpringBootApplication*

*@EnableBinding(Sink.class)*

**public** **class** XDeathApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.run(XDeathApplication.**class**, args);

}

*@StreamListener(Sink.INPUT)*

**public** **void** listen(String in, *@Header(name = "x-death", required = false)* Map<?,?> death) {

**if** (death != null && death.get("count").equals(3L)) {

*// giving up - don't send to DLX*

**throw** **new** ImmediateAcknowledgeAmqpException("Failed after 4 attempts");

}

**throw** **new** AmqpRejectAndDontRequeueException("failed");

}

}

Notice that the count property in the x-death header is a Long.

**39.5 Error Channels**

Starting with version 1.3, the binder unconditionally sends exceptions to an error channel for each consumer destination and can also be configured to send async producer send failures to an error channel. See “[???](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html)” for more information.

RabbitMQ has two types of send failures:

* Returned messages,
* Negatively acknowledged [Publisher Confirms](https://www.rabbitmq.com/confirms.html).

The latter is rare. According to the RabbitMQ documentation "[A nack] will only be delivered if an internal error occurs in the Erlang process responsible for a queue.".

As well as enabling producer error channels (as described in “[???](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html)”), the RabbitMQ binder only sends messages to the channels if the connection factory is appropriately configured, as follows:

* ccf.setPublisherConfirms(true);
* ccf.setPublisherReturns(true);

When using Spring Boot configuration for the connection factory, set the following properties:

* spring.rabbitmq.publisher-confirms
* spring.rabbitmq.publisher-returns

The payload of the ErrorMessage for a returned message is a ReturnedAmqpMessageException with the following properties:

* failedMessage: The spring-messaging Message<?> that failed to be sent.
* amqpMessage: The raw spring-amqp Message.
* replyCode: An integer value indicating the reason for the failure (for example, 312 - No route).
* replyText: A text value indicating the reason for the failure (for example, NO\_ROUTE).
* exchange: The exchange to which the message was published.
* routingKey: The routing key used when the message was published.

For negatively acknowledged confirmations, the payload is a NackedAmqpMessageException with the following properties:

* failedMessage: The spring-messaging Message<?> that failed to be sent.
* nackReason: A reason (if available — you may need to examine the broker logs for more information).

There is no automatic handling of these exceptions (such as sending to a [dead-letter queue](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#rabbit-dlq-processing)). You can consume these exceptions with your own Spring Integration flow.

**39.6 Dead-Letter Queue Processing**

Because you cannot anticipate how users would want to dispose of dead-lettered messages, the framework does not provide any standard mechanism to handle them. If the reason for the dead-lettering is transient, you may wish to route the messages back to the original queue. However, if the problem is a permanent issue, that could cause an infinite loop. The following Spring Boot application shows an example of how to route those messages back to the original queue but moves them to a third “parking lot” queue after three attempts. The second example uses the [RabbitMQ Delayed Message Exchange](https://www.rabbitmq.com/blog/2015/04/16/scheduling-messages-with-rabbitmq/) to introduce a delay to the re-queued message. In this example, the delay increases for each attempt. These examples use a @RabbitListener to receive messages from the DLQ. You could also use RabbitTemplate.receive() in a batch process.

The examples assume the original destination is so8400in and the consumer group is so8400.

**39.6.1 Non-Partitioned Destinations**

The first two examples are for when the destination is **not** partitioned:

*@SpringBootApplication*

**public** **class** ReRouteDlqApplication {

**private** **static** **final** String ORIGINAL\_QUEUE = "so8400in.so8400";

**private** **static** **final** String DLQ = ORIGINAL\_QUEUE + ".dlq";

**private** **static** **final** String PARKING\_LOT = ORIGINAL\_QUEUE + ".parkingLot";

**private** **static** **final** String X\_RETRIES\_HEADER = "x-retries";

**public** **static** **void** main(String[] args) **throws** Exception {

ConfigurableApplicationContext context = SpringApplication.run(ReRouteDlqApplication.**class**, args);

System.out.println("Hit enter to terminate");

System.in.read();

context.close();

}

*@Autowired*

**private** RabbitTemplate rabbitTemplate;

*@RabbitListener(queues = DLQ)*

**public** **void** rePublish(Message failedMessage) {

Integer retriesHeader = (Integer) failedMessage.getMessageProperties().getHeaders().get(X\_RETRIES\_HEADER);

**if** (retriesHeader == null) {

retriesHeader = Integer.valueOf(0);

}

**if** (retriesHeader < 3) {

failedMessage.getMessageProperties().getHeaders().put(X\_RETRIES\_HEADER, retriesHeader + 1);

**this**.rabbitTemplate.send(ORIGINAL\_QUEUE, failedMessage);

}

**else** {

**this**.rabbitTemplate.send(PARKING\_LOT, failedMessage);

}

}

*@Bean*

**public** Queue parkingLot() {

**return** **new** Queue(PARKING\_LOT);

}

}

*@SpringBootApplication*

**public** **class** ReRouteDlqApplication {

**private** **static** **final** String ORIGINAL\_QUEUE = "so8400in.so8400";

**private** **static** **final** String DLQ = ORIGINAL\_QUEUE + ".dlq";

**private** **static** **final** String PARKING\_LOT = ORIGINAL\_QUEUE + ".parkingLot";

**private** **static** **final** String X\_RETRIES\_HEADER = "x-retries";

**private** **static** **final** String DELAY\_EXCHANGE = "dlqReRouter";

**public** **static** **void** main(String[] args) **throws** Exception {

ConfigurableApplicationContext context = SpringApplication.run(ReRouteDlqApplication.**class**, args);

System.out.println("Hit enter to terminate");

System.in.read();

context.close();

}

*@Autowired*

**private** RabbitTemplate rabbitTemplate;

*@RabbitListener(queues = DLQ)*

**public** **void** rePublish(Message failedMessage) {

Map<String, Object> headers = failedMessage.getMessageProperties().getHeaders();

Integer retriesHeader = (Integer) headers.get(X\_RETRIES\_HEADER);

**if** (retriesHeader == null) {

retriesHeader = Integer.valueOf(0);

}

**if** (retriesHeader < 3) {

headers.put(X\_RETRIES\_HEADER, retriesHeader + 1);

headers.put("x-delay", 5000 \* retriesHeader);

**this**.rabbitTemplate.send(DELAY\_EXCHANGE, ORIGINAL\_QUEUE, failedMessage);

}

**else** {

**this**.rabbitTemplate.send(PARKING\_LOT, failedMessage);

}

}

*@Bean*

**public** DirectExchange delayExchange() {

DirectExchange exchange = **new** DirectExchange(DELAY\_EXCHANGE);

exchange.setDelayed(true);

**return** exchange;

}

*@Bean*

**public** Binding bindOriginalToDelay() {

**return** BindingBuilder.bind(**new** Queue(ORIGINAL\_QUEUE)).to(delayExchange()).with(ORIGINAL\_QUEUE);

}

*@Bean*

**public** Queue parkingLot() {

**return** **new** Queue(PARKING\_LOT);

}

}

**39.6.2 Partitioned Destinations**

With partitioned destinations, there is one DLQ for all partitions. We determine the original queue from the headers.

**republishToDlq=false**

When republishToDlq is false, RabbitMQ publishes the message to the DLX/DLQ with an x-death header containing information about the original destination, as shown in the following example:

*@SpringBootApplication*

**public** **class** ReRouteDlqApplication {

**private** **static** **final** String ORIGINAL\_QUEUE = "so8400in.so8400";

**private** **static** **final** String DLQ = ORIGINAL\_QUEUE + ".dlq";

**private** **static** **final** String PARKING\_LOT = ORIGINAL\_QUEUE + ".parkingLot";

**private** **static** **final** String X\_DEATH\_HEADER = "x-death";

**private** **static** **final** String X\_RETRIES\_HEADER = "x-retries";

**public** **static** **void** main(String[] args) **throws** Exception {

ConfigurableApplicationContext context = SpringApplication.run(ReRouteDlqApplication.**class**, args);

System.out.println("Hit enter to terminate");

System.in.read();

context.close();

}

*@Autowired*

**private** RabbitTemplate rabbitTemplate;

*@SuppressWarnings("unchecked")*

*@RabbitListener(queues = DLQ)*

**public** **void** rePublish(Message failedMessage) {

Map<String, Object> headers = failedMessage.getMessageProperties().getHeaders();

Integer retriesHeader = (Integer) headers.get(X\_RETRIES\_HEADER);

**if** (retriesHeader == null) {

retriesHeader = Integer.valueOf(0);

}

**if** (retriesHeader < 3) {

headers.put(X\_RETRIES\_HEADER, retriesHeader + 1);

List<Map<String, ?>> xDeath = (List<Map<String, ?>>) headers.get(X\_DEATH\_HEADER);

String exchange = (String) xDeath.get(0).get("exchange");

List<String> routingKeys = (List<String>) xDeath.get(0).get("routing-keys");

**this**.rabbitTemplate.send(exchange, routingKeys.get(0), failedMessage);

}

**else** {

**this**.rabbitTemplate.send(PARKING\_LOT, failedMessage);

}

}

*@Bean*

**public** Queue parkingLot() {

**return** **new** Queue(PARKING\_LOT);

}

}

**republishToDlq=true**

When republishToDlq is true, the republishing recoverer adds the original exchange and routing key to headers, as shown in the following example:

*@SpringBootApplication*

**public** **class** ReRouteDlqApplication {

**private** **static** **final** String ORIGINAL\_QUEUE = "so8400in.so8400";

**private** **static** **final** String DLQ = ORIGINAL\_QUEUE + ".dlq";

**private** **static** **final** String PARKING\_LOT = ORIGINAL\_QUEUE + ".parkingLot";

**private** **static** **final** String X\_RETRIES\_HEADER = "x-retries";

**private** **static** **final** String X\_ORIGINAL\_EXCHANGE\_HEADER = RepublishMessageRecoverer.X\_ORIGINAL\_EXCHANGE;

**private** **static** **final** String X\_ORIGINAL\_ROUTING\_KEY\_HEADER = RepublishMessageRecoverer.X\_ORIGINAL\_ROUTING\_KEY;

**public** **static** **void** main(String[] args) **throws** Exception {

ConfigurableApplicationContext context = SpringApplication.run(ReRouteDlqApplication.**class**, args);

System.out.println("Hit enter to terminate");

System.in.read();

context.close();

}

*@Autowired*

**private** RabbitTemplate rabbitTemplate;

*@RabbitListener(queues = DLQ)*

**public** **void** rePublish(Message failedMessage) {

Map<String, Object> headers = failedMessage.getMessageProperties().getHeaders();

Integer retriesHeader = (Integer) headers.get(X\_RETRIES\_HEADER);

**if** (retriesHeader == null) {

retriesHeader = Integer.valueOf(0);

}

**if** (retriesHeader < 3) {

headers.put(X\_RETRIES\_HEADER, retriesHeader + 1);

String exchange = (String) headers.get(X\_ORIGINAL\_EXCHANGE\_HEADER);

String originalRoutingKey = (String) headers.get(X\_ORIGINAL\_ROUTING\_KEY\_HEADER);

**this**.rabbitTemplate.send(exchange, originalRoutingKey, failedMessage);

}

**else** {

**this**.rabbitTemplate.send(PARKING\_LOT, failedMessage);

}

}

*@Bean*

**public** Queue parkingLot() {

**return** **new** Queue(PARKING\_LOT);

}

}

**39.7 Partitioning with the RabbitMQ Binder**

RabbitMQ does not support partitioning natively.

Sometimes, it is advantageous to send data to specific partitions — for example, when you want to strictly order message processing, all messages for a particular customer should go to the same partition.

The RabbitMessageChannelBinder provides partitioning by binding a queue for each partition to the destination exchange.

The following Java and YAML examples show how to configure the producer:

**Producer.**

*@SpringBootApplication*

*@EnableBinding(Source.class)*

**public** **class** RabbitPartitionProducerApplication {

**private** **static** **final** Random RANDOM = **new** Random(System.currentTimeMillis());

**private** **static** **final** String[] data = **new** String[] {

"abc1", "def1", "qux1",

"abc2", "def2", "qux2",

"abc3", "def3", "qux3",

"abc4", "def4", "qux4",

};

**public** **static** **void** main(String[] args) {

**new** SpringApplicationBuilder(RabbitPartitionProducerApplication.**class**)

.web(false)

.run(args);

}

*@InboundChannelAdapter(channel = Source.OUTPUT, poller = @Poller(fixedRate = "5000"))*

**public** Message<?> generate() {

String value = data[RANDOM.nextInt(data.length)];

System.out.println("Sending: " + value);

**return** MessageBuilder.withPayload(value)

.setHeader("partitionKey", value)

.build();

}

}

**application.yml.**

spring:

cloud:

stream:

bindings:

output:

destination: partitioned.destination

producer:

partitioned: **true**

partition-key-expression: headers['partitionKey'**]**

partition-count: 2

required-groups:

- myGroup

|  |
| --- |
| [Note] |
| The configuration in the prececing example uses the default partitioning (key.hashCode() % partitionCount). This may or may not provide a suitably balanced algorithm, depending on the key values. You can override this default by using the partitionSelectorExpression or partitionSelectorClass properties.  The required-groups property is required only if you need the consumer queues to be provisioned when the producer is deployed. Otherwise, any messages sent to a partition are lost until the corresponding consumer is deployed. |

The following configuration provisions a topic exchange:

The following queues are bound to that exchange:

The following bindings associate the queues to the exchange:

The following Java and YAML examples continue the previous examples and show how to configure the consumer:

**Consumer.**

*@SpringBootApplication*

*@EnableBinding(Sink.class)*

**public** **class** RabbitPartitionConsumerApplication {

**public** **static** **void** main(String[] args) {

**new** SpringApplicationBuilder(RabbitPartitionConsumerApplication.**class**)

.web(false)

.run(args);

}

*@StreamListener(Sink.INPUT)*

**public** **void** listen(*@Payload* String in, *@Header(AmqpHeaders.CONSUMER\_QUEUE)* String queue) {

System.out.println(in + " received from queue " + queue);

}

}

**application.yml.**

spring:

cloud:

stream:

bindings:

input:

destination: partitioned.destination

group: myGroup

consumer:

partitioned: **true**

instance-index: 0

|  |  |
| --- | --- |
| [Important] | **Important** |
| The RabbitMessageChannelBinder does not support dynamic scaling. There must be at least one consumer per partition. The consumer’s instanceIndex is used to indicate which partition is consumed. Platforms such as Cloud Foundry can have only one instance with an instanceIndex. |

**Part VII. Spring Cloud Bus**

Spring Cloud Bus links the nodes of a distributed system with a lightweight message broker. This broker can then be used to broadcast state changes (such as configuration changes) or other management instructions. A key idea is that the bus is like a distributed actuator for a Spring Boot application that is scaled out. However, it can also be used as a communication channel between apps. This project provides starters for either an AMQP broker or Kafka as the transport.

|  |
| --- |
| [Note] |
| Spring Cloud is released under the non-restrictive Apache 2.0 license. If you would like to contribute to this section of the documentation or if you find an error, please find the source code and issue trackers in the project at [github](https://github.com/spring-cloud/spring-cloud-config/tree/master/docs/src/main/asciidoc). |

**40. Quick Start**

Spring Cloud Bus works by adding Spring Boot autconfiguration if it detects itself on the classpath. To enable the bus, add spring-cloud-starter-bus-amqp orspring-cloud-starter-bus-kafka to your dependency management. Spring Cloud takes care of the rest. Make sure the broker (RabbitMQ or Kafka) is available and configured. When running on localhost, you need not do anything. If you run remotely, use Spring Cloud Connectors or Spring Boot conventions to define the broker credentials, as shown in the following example for Rabbit:

**application.yml.**

spring:

rabbitmq:

host: mybroker.com

port: 5672

username: user

password: secret

The bus currently supports sending messages to all nodes listening or all nodes for a particular service (as defined by Eureka). The /bus/\* actuator namespace has some HTTP endpoints. Currently, two are implemented. The first, /bus/env, sends key/value pairs to update each node’s Spring Environment. The second, /bus/refresh, reloads each application’s configuration, as though they had all been pinged on their /refresh endpoint.

|  |
| --- |
| [Note] |
| The Spring Cloud Bus starters cover Rabbit and Kafka, because those are the two most common implementations. However, Spring Cloud Stream is quite flexible, and the binder works with spring-cloud-bus. |

**41. Bus Endpoints**

Spring Cloud Bus provides two endpoints, /actuator/bus-refresh and /actuator/bus-env that correspond to individual actuator endpoints in Spring Cloud Commons, /actuator/refresh and /actuator/env respectively.

**41.1 Bus Refresh Endpoint**

The /actuator/bus-refresh endpoint clears the RefreshScope cache and rebinds @ConfigurationProperties. See the [Refresh Scope](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#refresh-scope) documentation for more information.

To expose the /actuator/bus-refresh endpoint, you need to add following configuration to your application:

management.endpoints.web.exposure.include=bus-refresh

**41.2 Bus Env Endpoint**

The /actuator/bus-env endpoint updates each instances environment with the specified key/value pair across multiple instances.

To expose the /actuator/bus-env endpoint, you need to add following configuration to your application:

management.endpoints.web.exposure.include=bus-env

The /actuator/bus-env endpoint accepts POST requests with the following shape:

**{**

"name": "key1"**,**

"value": "value1"

**}**

**42. Addressing an Instance**

Each instance of the application has a service ID, whose value can be set with spring.cloud.bus.id and whose value is expected to be a colon-separated list of identifiers, in order from least specific to most specific. The default value is constructed from the environment as a combination of the spring.application.name andserver.port (or spring.application.index, if set). The default value of the ID is constructed in the form of app:index:id, where:

* app is the vcap.application.name, if it exists, or spring.application.name
* index is the vcap.application.instance\_index, if it exists, spring.application.index, local.server.port, server.port, or 0 (in that order).
* id is the vcap.application.instance\_id, if it exists, or a random value.

The HTTP endpoints accept a “destination” path parameter, such as /bus-refresh/customers:9000, where destination is a service ID. If the ID is owned by an instance on the bus, it processes the message, and all other instances ignore it.

**43. Addressing All Instances of a Service**

The “destination” parameter is used in a Spring PathMatcher (with the path separator as a colon — :) to determine if an instance processes the message. Using the example from earlier, /bus-env/customers:\*\* targets all instances of the “customers” service regardless of the rest of the service ID.

**44. Service ID Must Be Unique**

The bus tries twice to eliminate processing an event — once from the original ApplicationEvent and once from the queue. To do so, it checks the sending service ID against the current service ID. If multiple instances of a service have the same ID, events are not processed. When running on a local machine, each service is on a different port, and that port is part of the ID. Cloud Foundry supplies an index to differentiate. To ensure that the ID is unique outside Cloud Foundry, set spring.application.index to something unique for each instance of a service.

**45. Customizing the Message Broker**

Spring Cloud Bus uses [Spring Cloud Stream](https://cloud.spring.io/spring-cloud-stream) to broadcast the messages. So, to get messages to flow, you need only include the binder implementation of your choice in the classpath. There are convenient starters for the bus with AMQP (RabbitMQ) and Kafka (spring-cloud-starter-bus-[amqp|kafka]). Generally speaking, Spring Cloud Stream relies on Spring Boot autoconfiguration conventions for configuring middleware. For instance, the AMQP broker address can be changed withspring.rabbitmq.\* configuration properties. Spring Cloud Bus has a handful of native configuration properties in spring.cloud.bus.\* (for example,spring.cloud.bus.destination is the name of the topic to use as the external middleware). Normally, the defaults suffice.

To learn more about how to customize the message broker settings, consult the Spring Cloud Stream documentation.

**46. Tracing Bus Events**

Bus events (subclasses of RemoteApplicationEvent) can be traced by setting spring.cloud.bus.trace.enabled=true. If you do so, the Spring Boot TraceRepository (if it is present) shows each event sent and all the acks from each service instance. The following example comes from the /trace endpoint:

**{**

"timestamp": "2015-11-26T10:24:44.411+0000"**,**

"info": **{**

"signal": "spring.cloud.bus.ack"**,**

"type": "RefreshRemoteApplicationEvent"**,**

"id": "c4d374b7-58ea-4928-a312-31984def293b"**,**

"origin": "stores:8081"**,**

"destination": "\*:\*\*"

**}**

**},**

**{**

"timestamp": "2015-11-26T10:24:41.864+0000"**,**

"info": **{**

"signal": "spring.cloud.bus.sent"**,**

"type": "RefreshRemoteApplicationEvent"**,**

"id": "c4d374b7-58ea-4928-a312-31984def293b"**,**

"origin": "customers:9000"**,**

"destination": "\*:\*\*"

**}**

**},**

**{**

"timestamp": "2015-11-26T10:24:41.862+0000"**,**

"info": **{**

"signal": "spring.cloud.bus.ack"**,**

"type": "RefreshRemoteApplicationEvent"**,**

"id": "c4d374b7-58ea-4928-a312-31984def293b"**,**

"origin": "customers:9000"**,**

"destination": "\*:\*\*"

**}**

**}**

The preceding trace shows that a RefreshRemoteApplicationEvent was sent from customers:9000, broadcast to all services, and received (acked) by customers:9000 and stores:8081.

To handle the ack signals yourself, you could add an @EventListener for the AckRemoteApplicationEvent and SentApplicationEvent types to your app (and enable tracing). Alternatively, you could tap into the TraceRepository and mine the data from there.

|  |
| --- |
| [Note] |
| Any Bus application can trace acks. However, sometimes, it is useful to do this in a central service that can do more complex queries on the data or forward it to a specialized tracing service. |

**47. Broadcasting Your Own Events**

The Bus can carry any event of type RemoteApplicationEvent. The default transport is JSON, and the deserializer needs to know which types are going to be used ahead of time. To register a new type, you must put it in a subpackage of org.springframework.cloud.bus.event.

To customise the event name, you can use @JsonTypeName on your custom class or rely on the default strategy, which is to use the simple name of the class.

|  |
| --- |
| [Note] |
| Both the producer and the consumer need access to the class definition. |

**47.1 Registering events in custom packages**

If you cannot or do not want to use a subpackage of org.springframework.cloud.bus.event for your custom events, you must specify which packages to scan for events of type RemoteApplicationEvent by using the @RemoteApplicationEventScan annotation. Packages specified with @RemoteApplicationEventScaninclude subpackages.

For example, consider the following custom event, called MyEvent:

**package** com.acme;

**public** **class** MyEvent **extends** RemoteApplicationEvent {

...

}

You can register that event with the deserializer in the following way:

**package** com.acme;

*@Configuration*

*@RemoteApplicationEventScan*

**public** **class** BusConfiguration {

...

}

Without specifying a value, the package of the class where @RemoteApplicationEventScan is used is registered. In this example, com.acme is registered by using the package of BusConfiguration.

You can also explicitly specify the packages to scan by using the value, basePackages or basePackageClasses properties on @RemoteApplicationEventScan, as shown in the following example:

**package** com.acme;

*@Configuration*

*//@RemoteApplicationEventScan({"com.acme", "foo.bar"})*

*//@RemoteApplicationEventScan(basePackages = {"com.acme", "foo.bar", "fizz.buzz"})*

*@RemoteApplicationEventScan(basePackageClasses = BusConfiguration.class)*

**public** **class** BusConfiguration {

...

}

All of the preceding examples of @RemoteApplicationEventScan are equivalent, in that the com.acme package is registered by explicitly specifying the packages on@RemoteApplicationEventScan.

|  |
| --- |
| [Note] |
| You can specify multiple base packages to scan. |

**Part VIII. Spring Cloud Sleuth**

Adrian Cole, Spencer Gibb, Marcin Grzejszczak, Dave Syer, Jay Bryant

**Finchley.SR1**

**48. Introduction**

Spring Cloud Sleuth implements a distributed tracing solution for [Spring Cloud](https://cloud.spring.io/).

**48.1 Terminology**

Spring Cloud Sleuth borrows [Dapper’s](http://research.google.com/pubs/pub36356.html) terminology.

**Span**: The basic unit of work. For example, sending an RPC is a new span, as is sending a response to an RPC. Spans are identified by a unique 64-bit ID for the span and another 64-bit ID for the trace the span is a part of. Spans also have other data, such as descriptions, timestamped events, key-value annotations (tags), the ID of the span that caused them, and process IDs (normally IP addresses).

Spans can be started and stopped, and they keep track of their timing information. Once you create a span, you must stop it at some point in the future.

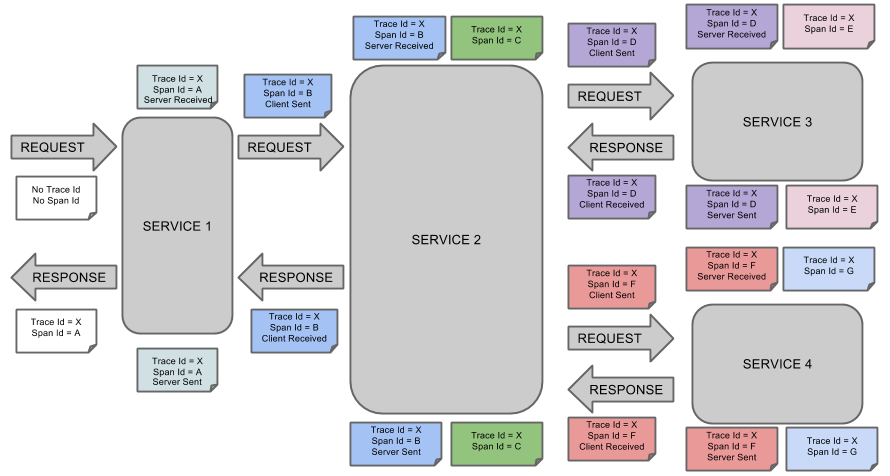
|  |
| --- |
| [Tip] |
| The initial span that starts a trace is called a root span. The value of the ID of that span is equal to the trace ID. |

**Trace:** A set of spans forming a tree-like structure. For example, if you run a distributed big-data store, a trace might be formed by a PUT request.

**Annotation:** Used to record the existence of an event in time. With [Brave](https://github.com/openzipkin/brave) instrumentation, we no longer need to set special events for [Zipkin](https://zipkin.io/) to understand who the client and server are, where the request started, and where it ended. For learning purposes, however, we mark these events to highlight what kind of an action took place.

* **cs**: Client Sent. The client has made a request. This annotation indicates the start of the span.
* **sr**: Server Received: The server side got the request and started processing it. Subtracting the cs timestamp from this timestamp reveals the network latency.
* **ss**: Server Sent. Annotated upon completion of request processing (when the response got sent back to the client). Subtracting the sr timestamp from this timestamp reveals the time needed by the server side to process the request.
* **cr**: Client Received. Signifies the end of the span. The client has successfully received the response from the server side. Subtracting the cs timestamp from this timestamp reveals the whole time needed by the client to receive the response from the server.

The following image shows how **Span** and **Trace** look in a system, together with the Zipkin annotations:



Each color of a note signifies a span (there are seven spans - from **A** to **G**). Consider the following note:

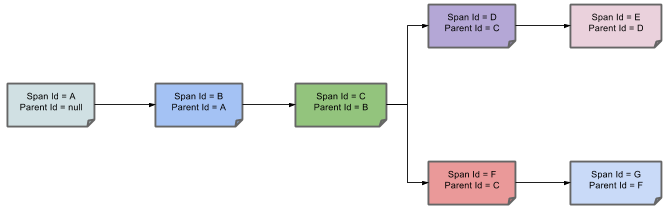
Trace Id = X

Span Id = D

Client Sent

This note indicates that the current span has **Trace Id** set to **X** and **Span Id** set to **D**. Also, the Client Sent event took place.

The following image shows how parent-child relationships of spans look:

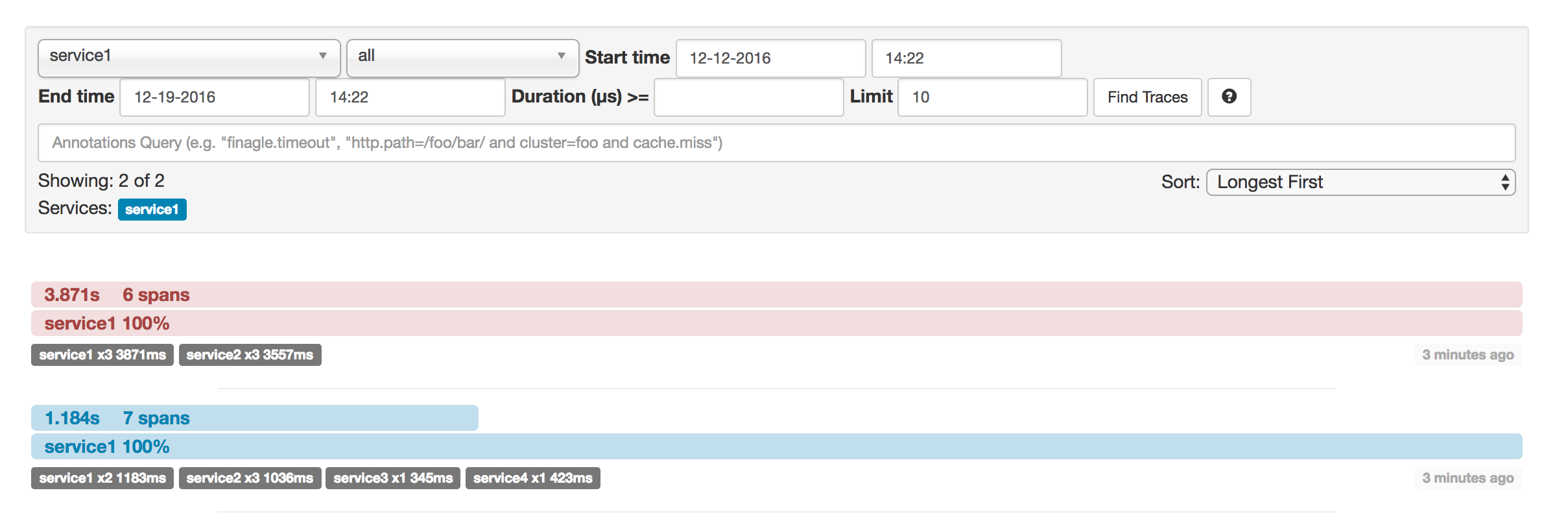


**48.2 Purpose**

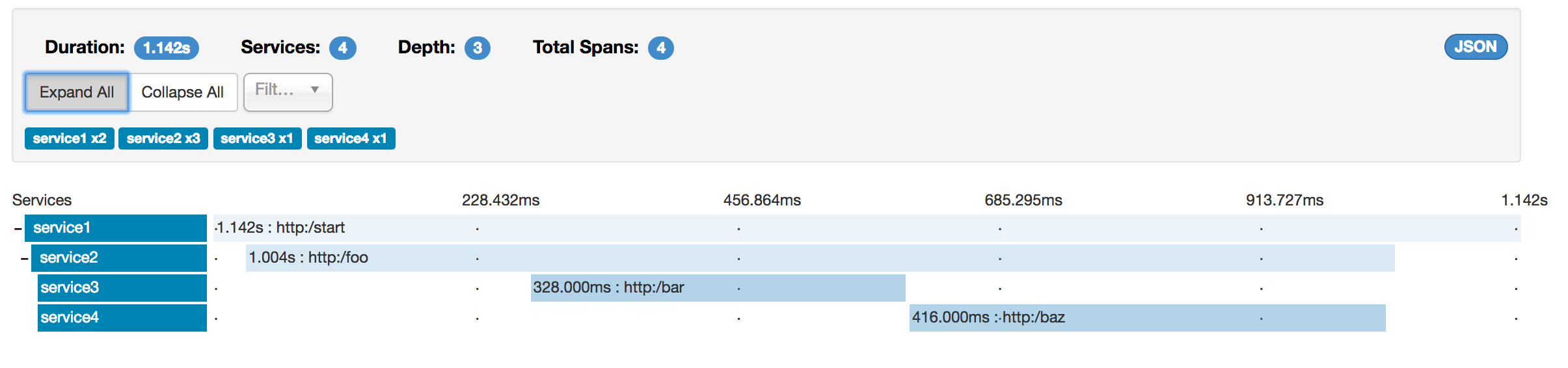
The following sections refer to the example shown in the preceding image.

**48.2.1 Distributed Tracing with Zipkin**

This example has seven spans. If you go to traces in Zipkin, you can see this number in the second trace, as shown in the following image:



However, if you pick a particular trace, you can see four spans, as shown in the following image:



|  |
| --- |
| [Note] |
| When you pick a particular trace, you see merged spans. That means that, if there were two spans sent to Zipkin with Server Received and Server Sent or Client Received and Client Sent annotations, they are presented as a single span. |

Why is there a difference between the seven and four spans in this case?

* Two spans come from the http:/start span. It has the Server Received (sr) and Server Sent (ss) annotations.
* Two spans come from the RPC call from service1 to service2 to the http:/foo endpoint. The Client Sent (cs) and Client Received (cr) events took place on the service1 side. Server Received (sr) and Server Sent (ss) events took place on the service2 side. These two spans form one logical span related to an RPC call.
* Two spans come from the RPC call from service2 to service3 to the http:/bar endpoint. The Client Sent (cs) and Client Received (cr) events took place on the service2 side. The Server Received (sr) and Server Sent (ss) events took place on the service3 side. These two spans form one logical span related to an RPC call.
* Two spans come from the RPC call from service2 to service4 to the http:/baz endpoint. The Client Sent (cs) and Client Received (cr) events took place on the service2 side. Server Received (sr) and Server Sent (ss) events took place on the service4 side. These two spans form one logical span related to an RPC call.

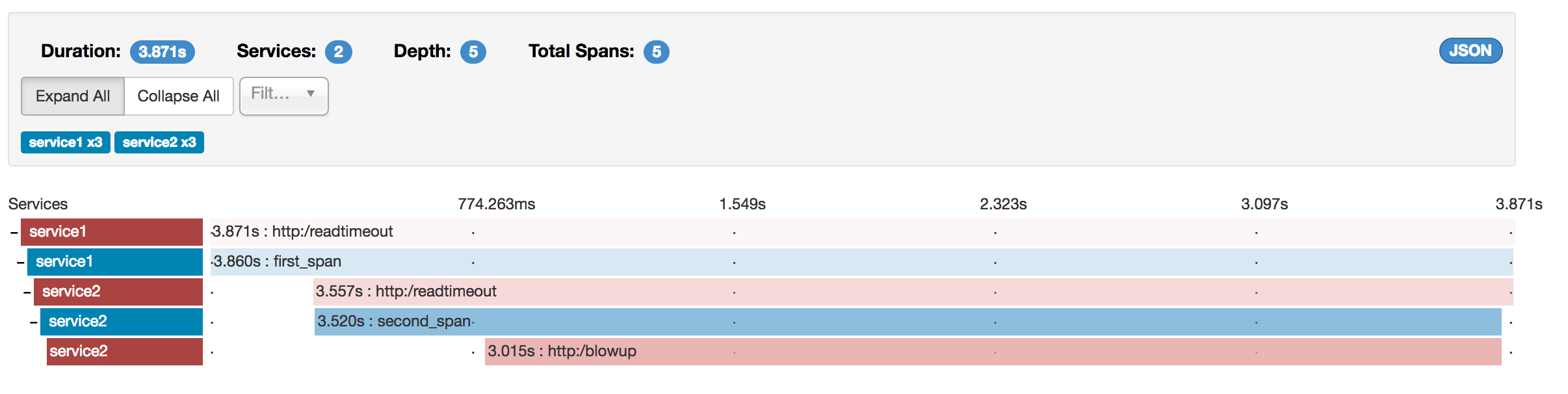
So, if we count the physical spans, we have one from http:/start, two from service1 calling service2, two from service2 calling service3, and two from service2 calling service4. In sum, we have a total of seven spans.

Logically, we see the information of four total Spans because we have one span related to the incoming request to service1 and three spans related to RPC calls.

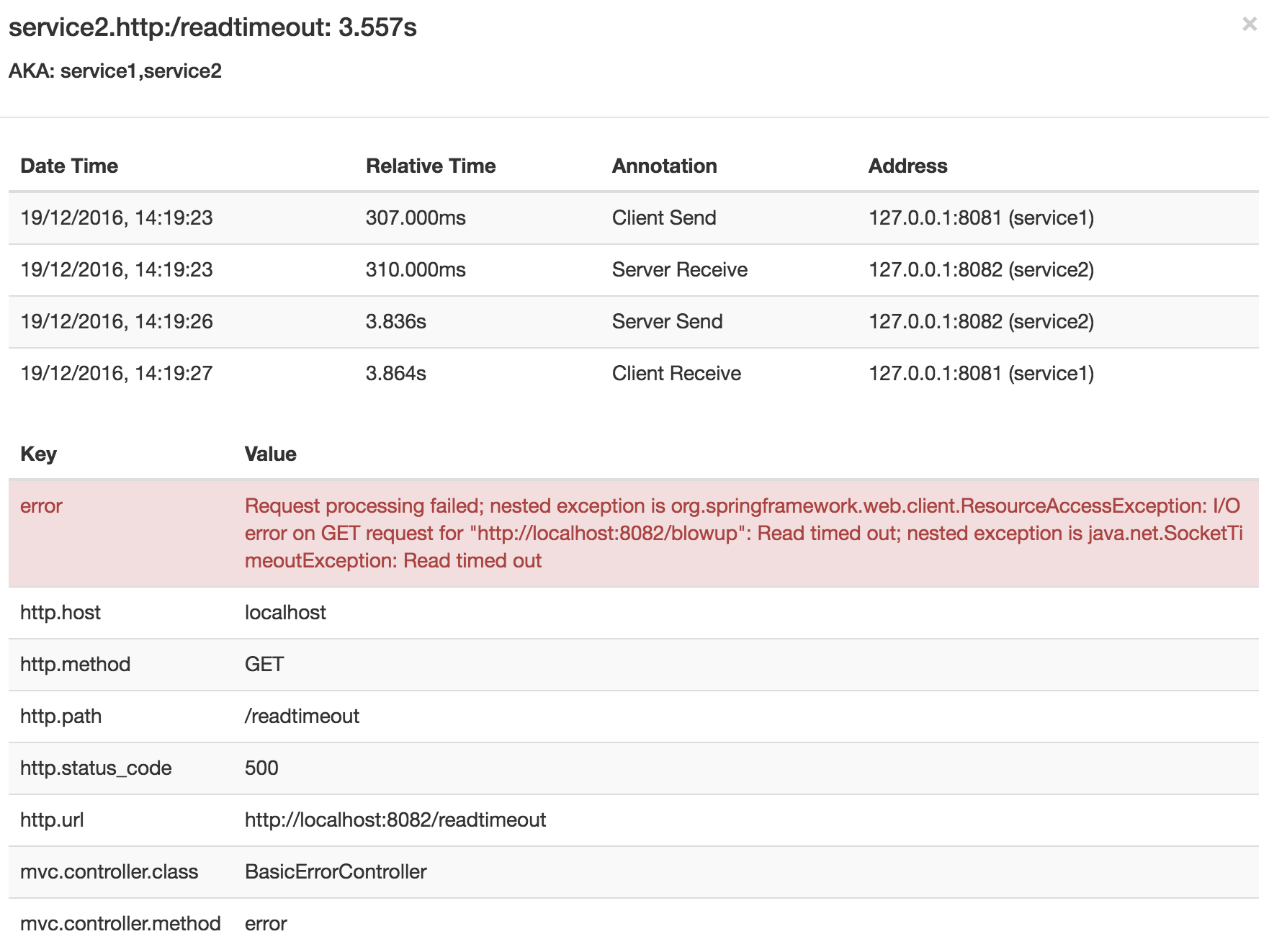
**48.2.2 Visualizing errors**

Zipkin lets you visualize errors in your trace. When an exception was thrown and was not caught, we set proper tags on the span, which Zipkin can then properly colorize. You could see in the list of traces one trace that is red. That appears because an exception was thrown.

If you click that trace, you see a similar picture, as follows:



If you then click on one of the spans, you see the following



The span shows the reason for the error and the whole stack trace related to it.

**48.2.3 Distributed Tracing with Brave**

Starting with version 2.0.0, Spring Cloud Sleuth uses [Brave](https://github.com/openzipkin/brave) as the tracing library. Consequently, Sleuth no longer takes care of storing the context but delegates that work to Brave.

Due to the fact that Sleuth had different naming and tagging conventions than Brave, we decided to follow Brave’s conventions from now on. However, if you want to use the legacy Sleuth approaches, you can set the spring.sleuth.http.legacy.enabled property to true.

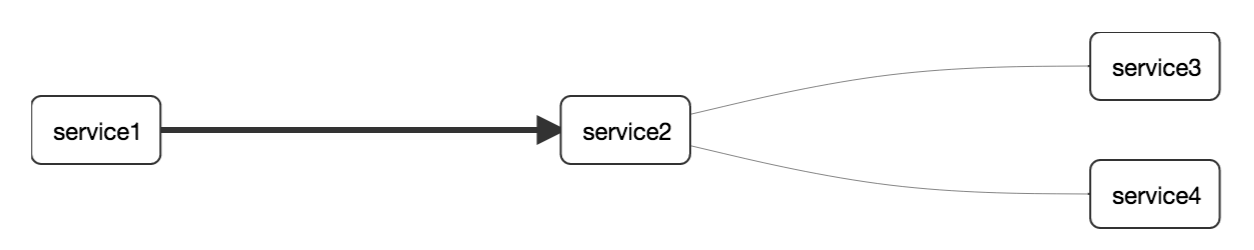
**48.2.4 Live examples**

**Figure 48.1. Click the Pivotal Web Services icon to see it live!**



[Click here to see it live!](https://docssleuth-zipkin-server.cfapps.io/)

The dependency graph in Zipkin should resemble the following image:



**Figure 48.2. Click the Pivotal Web Services icon to see it live!**



[Click here to see it live!](https://docssleuth-zipkin-server.cfapps.io/dependency)

**48.2.5 Log correlation**

When using grep to read the logs of those four applications by scanning for a trace ID equal to (for example) 2485ec27856c56f4, you get output resembling the following:

service1.log:2016-02-26 11:15:47.561 INFO [service1,2485ec27856c56f4,2485ec27856c56f4,true] 68058 --- [nio-8081-exec-1] i.s.c.sleuth.docs.service1.Application : Hello from service1. Calling service2

service2.log:2016-02-26 11:15:47.710 INFO [service2,2485ec27856c56f4,9aa10ee6fbde75fa,true] 68059 --- [nio-8082-exec-1] i.s.c.sleuth.docs.service2.Application : Hello from service2. Calling service3 and then service4

service3.log:2016-02-26 11:15:47.895 INFO [service3,2485ec27856c56f4,1210be13194bfe5,true] 68060 --- [nio-8083-exec-1] i.s.c.sleuth.docs.service3.Application : Hello from service3

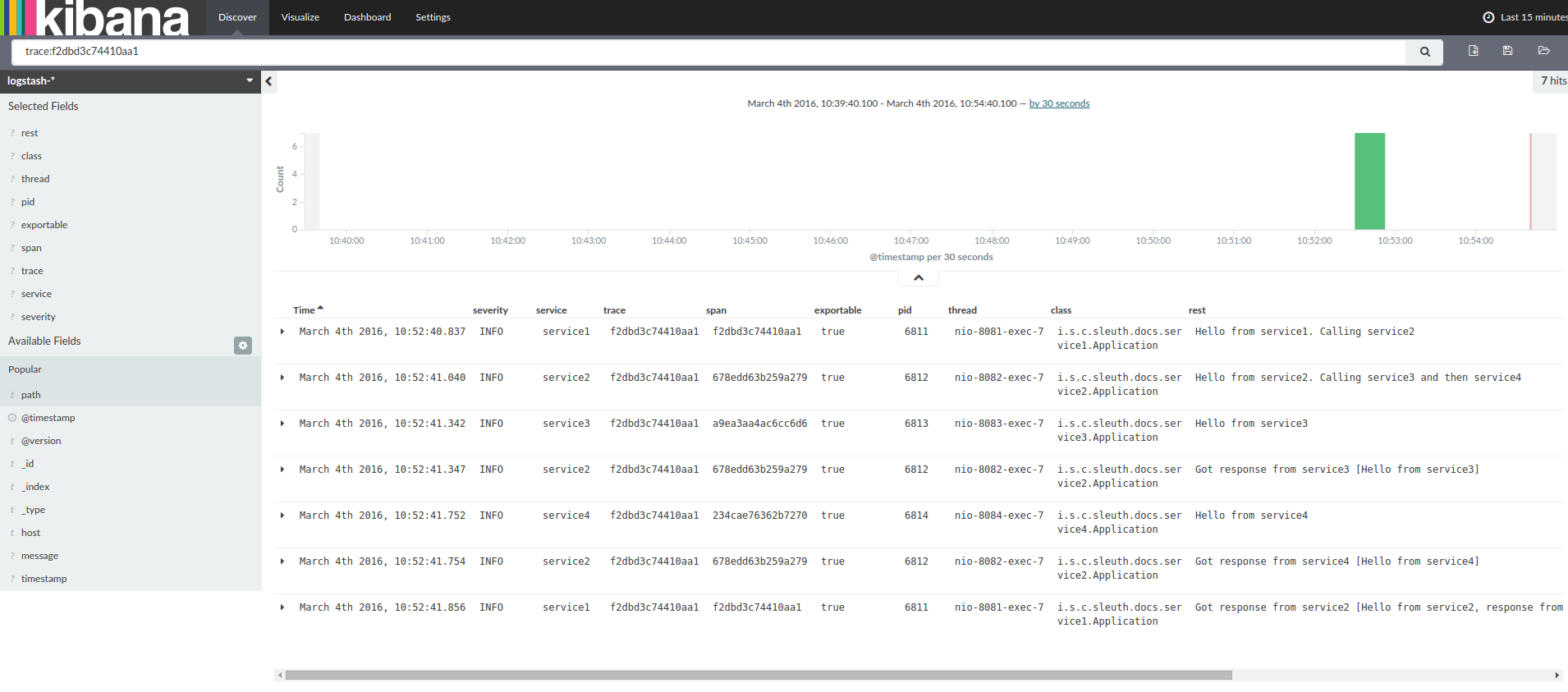
service2.log:2016-02-26 11:15:47.924 INFO [service2,2485ec27856c56f4,9aa10ee6fbde75fa,true] 68059 --- [nio-8082-exec-1] i.s.c.sleuth.docs.service2.Application : Got response from service3 [Hello from service3]

service4.log:2016-02-26 11:15:48.134 INFO [service4,2485ec27856c56f4,1b1845262ffba49d,true] 68061 --- [nio-8084-exec-1] i.s.c.sleuth.docs.service4.Application : Hello from service4

service2.log:2016-02-26 11:15:48.156 INFO [service2,2485ec27856c56f4,9aa10ee6fbde75fa,true] 68059 --- [nio-8082-exec-1] i.s.c.sleuth.docs.service2.Application : Got response from service4 [Hello from service4]

service1.log:2016-02-26 11:15:48.182 INFO [service1,2485ec27856c56f4,2485ec27856c56f4,true] 68058 --- [nio-8081-exec-1] i.s.c.sleuth.docs.service1.Application : Got response from service2 [Hello from service2, response from service3 [Hello from service3] and from service4 [Hello from service4]]

If you use a log aggregating tool (such as [Kibana](https://www.elastic.co/products/kibana), [Splunk](http://www.splunk.com/), and others), you can order the events that took place. An example from Kibana would resemble the following image:



If you want to use [Logstash](https://www.elastic.co/guide/en/logstash/current/index.html), the following listing shows the Grok pattern for Logstash:

filter {

# pattern matching logback pattern

grok {

match => { "message" => "%{TIMESTAMP\_ISO8601:timestamp}\s+%{LOGLEVEL:severity}\s+\[%{DATA:service},%{DATA:trace},%{DATA:span},%{DATA:exportable}\]\s+%{DATA:pid}\s+---\s+\[%{DATA:thread}\]\s+%{DATA:class}\s+:\s+%{GREEDYDATA:rest}" }

}

}

|  |
| --- |
| [Note] |
| If you want to use Grok together with the logs from Cloud Foundry, you have to use the following pattern: |

filter {

# pattern matching logback pattern

grok {

match => { "message" => "(?m)OUT\s+%{TIMESTAMP\_ISO8601:timestamp}\s+%{LOGLEVEL:severity}\s+\[%{DATA:service},%{DATA:trace},%{DATA:span},%{DATA:exportable}\]\s+%{DATA:pid}\s+---\s+\[%{DATA:thread}\]\s+%{DATA:class}\s+:\s+%{GREEDYDATA:rest}" }

}

}

**JSON Logback with Logstash**

Often, you do not want to store your logs in a text file but in a JSON file that Logstash can immediately pick. To do so, you have to do the following (for readability, we pass the dependencies in the groupId:artifactId:version notation).

**Dependencies Setup**

1. Ensure that Logback is on the classpath (ch.qos.logback:logback-core).
2. Add Logstash Logback encode. For example, to use version 4.6, add net.logstash.logback:logstash-logback-encoder:4.6.

**Logback Setup**

Consider the following example of a Logback configuration file (named [logback-spring.xml](https://github.com/spring-cloud-samples/sleuth-documentation-apps/blob/master/service1/src/main/resources/logback-spring.xml)).

<?xml version="1.0" encoding="UTF-8"?>

<configuration>

<include resource="org/springframework/boot/logging/logback/defaults.xml"/>

​

<springProperty scope="context" name="springAppName" source="spring.application.name"/>

*<!-- Example for logging into the build folder of your project -->*

<property name="LOG\_FILE" value="${BUILD\_FOLDER:-build}/${springAppName}"/>​

*<!-- You can override this to have a custom pattern -->*

<property name="CONSOLE\_LOG\_PATTERN"

value="%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){faint} %clr(${LOG\_LEVEL\_PATTERN:-%5p}) %clr(${PID:- }){magenta} %clr(---){faint} %clr([%15.15t]){faint} %clr(%-40.40logger{39}){cyan} %clr(:){faint} %m%n${LOG\_EXCEPTION\_CONVERSION\_WORD:-%wEx}"/>

*<!-- Appender to log to console -->*

<appender name="console" class="ch.qos.logback.core.ConsoleAppender">

<filter class="ch.qos.logback.classic.filter.ThresholdFilter">

*<!-- Minimum logging level to be presented in the console logs-->*

<level>DEBUG</level>

</filter>

<encoder>

<pattern>${CONSOLE\_LOG\_PATTERN}</pattern>

<charset>utf8</charset>

</encoder>

</appender>

*<!-- Appender to log to file -->*​

<appender name="flatfile" class="ch.qos.logback.core.rolling.RollingFileAppender">

<file>${LOG\_FILE}</file>

<rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">

<fileNamePattern>${LOG\_FILE}.%d{yyyy-MM-dd}.gz</fileNamePattern>

<maxHistory>7</maxHistory>

</rollingPolicy>

<encoder>

<pattern>${CONSOLE\_LOG\_PATTERN}</pattern>

<charset>utf8</charset>

</encoder>

</appender>

​

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

<appender name="logstash" class="ch.qos.logback.core.rolling.RollingFileAppender">

<file>${LOG\_FILE}.json</file>

<rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">

<fileNamePattern>${LOG\_FILE}.json.%d{yyyy-MM-dd}.gz</fileNamePattern>

<maxHistory>7</maxHistory>

</rollingPolicy>

<encoder class="net.logstash.logback.encoder.LoggingEventCompositeJsonEncoder">

<providers>

<timestamp>

<timeZone>UTC</timeZone>

</timestamp>

<pattern>

<pattern>

{

"severity": "%level",

"service": "${springAppName:-}",

"trace": "%X{X-B3-TraceId:-}",

"span": "%X{X-B3-SpanId:-}",

"parent": "%X{X-B3-ParentSpanId:-}",

"exportable": "%X{X-Span-Export:-}",

"pid": "${PID:-}",

"thread": "%thread",

"class": "%logger{40}",

"rest": "%message"

}

</pattern>

</pattern>

</providers>

</encoder>

</appender>

​

<root level="INFO">

<appender-ref ref="console"/>

*<!-- uncomment this to have also JSON logs -->*

*<!--<appender-ref ref="logstash"/>-->*

*<!--<appender-ref ref="flatfile"/>-->*

</root>

</configuration>

That Logback configuration file:

* Logs information from the application in a JSON format to a build/${spring.application.name}.json file.
* Has commented out two additional appenders: console and standard log file.
* Has the same logging pattern as the one presented in the previous section.

|  |
| --- |
| [Note] |
| If you use a custom logback-spring.xml, you must pass the spring.application.name in the bootstrap rather than the application property file. Otherwise, your custom logback file does not properly read the property. |

**48.2.6 Propagating Span Context**

The span context is the state that must get propagated to any child spans across process boundaries. Part of the Span Context is the Baggage. The trace and span IDs are a required part of the span context. Baggage is an optional part.

Baggage is a set of key:value pairs stored in the span context. Baggage travels together with the trace and is attached to every span. Spring Cloud Sleuth understands that a header is baggage-related if the HTTP header is prefixed with baggage- and, for messaging, it starts with baggage\_.

|  |  |
| --- | --- |
| [Important] | **Important** |
| There is currently no limitation of the count or size of baggage items. However, keep in mind that too many can decrease system throughput or increase RPC latency. In extreme cases, too much baggage can crash the application, due to exceeding transport-level message or header capacity. |

The following example shows setting baggage on a span:

Span initialSpan = **this**.tracer.nextSpan().name("span").start();

ExtraFieldPropagation.set(initialSpan.context(), "foo", "bar");

ExtraFieldPropagation.set(initialSpan.context(),"UPPER\_CASE", "someValue");

}

**Baggage versus Span Tags**

Baggage travels with the trace (every child span contains the baggage of its parent). Zipkin has no knowledge of baggage and does not receive that information.

|  |  |
| --- | --- |
| [Important] | **Important** |
| Starting from Sleuth 2.0.0 you have to pass the baggage key names explicitly in your project configuration. Read more about that setup [here](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#prefixed-fields) |

Tags are attached to a specific span. In other words, they are presented only for that particular span. However, you can search by tag to find the trace, assuming a span having the searched tag value exists.

If you want to be able to lookup a span based on baggage, you should add a corresponding entry as a tag in the root span.

|  |  |
| --- | --- |
| [Important] | **Important** |
| The span must be in scope. |

The following listing shows integration tests that use baggage:

**The setup.**

spring.sleuth:

baggage-keys:

- baz

- bizarrecase

propagation-keys:

- foo

- upper\_case

**The code.**

initialSpan.tag("foo",

ExtraFieldPropagation.get(initialSpan.context(), "foo"));

initialSpan.tag("UPPER\_CASE",

ExtraFieldPropagation.get(initialSpan.context(), "UPPER\_CASE"));

**48.3 Adding Sleuth to the Project**

This section addresses how to add Sleuth to your project with either Maven or Gradle.

|  |  |
| --- | --- |
| [Important] | **Important** |
| To ensure that your application name is properly displayed in Zipkin, set the spring.application.name property in bootstrap.yml. |

**48.3.1 Only Sleuth (log correlation)**

If you want to use only Spring Cloud Sleuth without the Zipkin integration, add the spring-cloud-starter-sleuth module to your project.

The following example shows how to add Sleuth with Maven:

**Maven.**

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>${release.train.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-sleuth</artifactId>

</dependency>

|  |  |
| --- | --- |
|  | We recommend that you add the dependency management through the Spring BOM so that you need not manage versions yourself. |
|  | Add the dependency to spring-cloud-starter-sleuth. |

The following example shows how to add Sleuth with Gradle:

**Gradle.**

dependencyManagement {

imports {

mavenBom "org.springframework.cloud:spring-cloud-dependencies:${releaseTrainVersion}"

}

}

dependencies {

compile "org.springframework.cloud:spring-cloud-starter-sleuth"

}

|  |  |
| --- | --- |
|  | We recommend that you add the dependency management through the Spring BOM so that you need not manage versions yourself. |
|  | Add the dependency to spring-cloud-starter-sleuth. |

**48.3.2 Sleuth with Zipkin via HTTP**

If you want both Sleuth and Zipkin, add the spring-cloud-starter-zipkin dependency.

The following example shows how to do so for Maven:

**Maven.**

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>${release.train.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-zipkin</artifactId>

</dependency>

|  |  |
| --- | --- |
|  | We recommend that you add the dependency management through the Spring BOM so that you need not manage versions yourself. |
|  | Add the dependency to spring-cloud-starter-zipkin. |

The following example shows how to do so for Gradle:

**Gradle.**

dependencyManagement {

imports {

mavenBom "org.springframework.cloud:spring-cloud-dependencies:${releaseTrainVersion}"

}

}

dependencies {

compile "org.springframework.cloud:spring-cloud-starter-zipkin"

}

|  |  |
| --- | --- |
|  | We recommend that you add the dependency management through the Spring BOM so that you need not manage versions yourself. |
|  | Add the dependency to spring-cloud-starter-zipkin. |

**48.3.3 Sleuth with Zipkin over RabbitMQ or Kafka**

If you want to use RabbitMQ or Kafka instead of HTTP, add the spring-rabbit or spring-kafka dependency. The default destination name is zipkin.

If using Kafka, you must set the property spring.zipkin.sender.type property accordingly:

spring.zipkin.sender.type: kafka

|  |  |
| --- | --- |
| [Caution] | **Caution** |
| spring-cloud-sleuth-stream is deprecated and incompatible with these destinations. |

If you want Sleuth over RabbitMQ, add the spring-cloud-starter-zipkin and spring-rabbit dependencies.

The following example shows how to do so for Gradle:

**Maven.**

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>${release.train.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-zipkin</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.amqp</groupId>

<artifactId>spring-rabbit</artifactId>

</dependency>

|  |  |
| --- | --- |
|  | We recommend that you add the dependency management through the Spring BOM so that you need not manage versions yourself. |
|  | Add the dependency to spring-cloud-starter-zipkin. That way, all nested dependencies get downloaded. |
|  | To automatically configure RabbitMQ, add the spring-rabbit dependency. |

**Gradle.**

dependencyManagement {

imports {

mavenBom "org.springframework.cloud:spring-cloud-dependencies:${releaseTrainVersion}"

}

}

dependencies {

compile "org.springframework.cloud:spring-cloud-starter-zipkin"

compile "org.springframework.amqp:spring-rabbit"

}

|  |  |
| --- | --- |
|  | We recommend that you add the dependency management through the Spring BOM so that you need not manage versions yourself. |
|  | Add the dependency to spring-cloud-starter-zipkin. That way, all nested dependencies get downloaded. |
|  | To automatically configure RabbitMQ, add the spring-rabbit dependency. |

**49. Additional Resources**

You can watch a video of [Reshmi Krishna](https://twitter.com/reshmi9k) and [Marcin Grzejszczak](https://twitter.com/mgrzejszczak) talking about Spring Cloud Sleuth and Zipkin [by clicking here](https://content.pivotal.io/springone-platform-2017/distributed-tracing-latency-analysis-for-your-microservices-grzejszczak-krishna).

You can check different setups of Sleuth and Brave [in the openzipkin/sleuth-webmvc-example repository](https://github.com/openzipkin/sleuth-webmvc-example).

**50. Features**

* Adds trace and span IDs to the Slf4J MDC, so you can extract all the logs from a given trace or span in a log aggregator, as shown in the following example logs:
* 2016-02-02 15:30:57.902 INFO [bar,6bfd228dc00d216b,6bfd228dc00d216b,false] 23030 --- [nio-8081-exec-3] ...
* 2016-02-02 15:30:58.372 ERROR [bar,6bfd228dc00d216b,6bfd228dc00d216b,false] 23030 --- [nio-8081-exec-3] ...

2016-02-02 15:31:01.936 INFO [bar,46ab0d418373cbc9,46ab0d418373cbc9,false] 23030 --- [nio-8081-exec-4] ...

Notice the [appname,traceId,spanId,exportable] entries from the MDC:

* + **spanId**: The ID of a specific operation that took place.
  + **appname**: The name of the application that logged the span.
  + **traceId**: The ID of the latency graph that contains the span.
  + **exportable**: Whether the log should be exported to Zipkin. When would you like the span not to be exportable? When you want to wrap some operation in a Span and have it written to the logs only.
* Provides an abstraction over common distributed tracing data models: traces, spans (forming a DAG), annotations, and key-value annotations. Spring Cloud Sleuth is loosely based on HTrace but is compatible with Zipkin (Dapper).
* Sleuth records timing information to aid in latency analysis. By using sleuth, you can pinpoint causes of latency in your applications.
* Sleuth is written to not log too much and to not cause your production application to crash. To that end, Sleuth:
  + Propagates structural data about your call graph in-band and the rest out-of-band.
  + Includes opinionated instrumentation of layers such as HTTP.
  + Includes a sampling policy to manage volume.
  + Can report to a Zipkin system for query and visualization.
* Instruments common ingress and egress points from Spring applications (servlet filter, async endpoints, rest template, scheduled actions, message channels, Zuul filters, and Feign client).
* Sleuth includes default logic to join a trace across HTTP or messaging boundaries. For example, HTTP propagation works over Zipkin-compatible request headers.
* Sleuth can propagate context (also known as baggage) between processes. Consequently, if you set a baggage element on a Span, it is sent downstream to other processes over either HTTP or messaging.
* Provides a way to create or continue spans and add tags and logs through annotations.
* If spring-cloud-sleuth-zipkin is on the classpath, the app generates and collects Zipkin-compatible traces. By default, it sends them over HTTP to a Zipkin server on localhost (port 9411). You can configure the location of the service by setting spring.zipkin.baseUrl.
  + If you depend on spring-rabbit, your app sends traces to a RabbitMQ broker instead of HTTP.
  + If you depend on spring-kafka, and set spring.zipkin.sender.type: kafka, your app sends traces to a Kafka broker instead of HTTP.

|  |  |
| --- | --- |
| [Caution] | **Caution** |
| spring-cloud-sleuth-stream is deprecated and should no longer be used. |

* Spring Cloud Sleuth is [OpenTracing](http://opentracing.io/) compatible.

|  |  |  |
| --- | --- | --- |
| [Important] | | **Important** |
| If you use Zipkin, configure the probability of spans exported by setting spring.sleuth.sampler.probability (default: 0.1, which is 10 percent). Otherwise, you might think that Sleuth is not working be cause it omits some spans. |
| [Note] |
| The SLF4J MDC is always set and logback users immediately see the trace and span IDs in logs per the example shown earlier. Other logging systems have to configure their own formatter to get the same result. The default is as follows: logging.pattern.level set to %5p [${spring.zipkin.service.name:${spring.application.name:-}},%X{X-B3-TraceId:-},%X{X-B3-SpanId:-},%X{X-Span-Export:-}](this is a Spring Boot feature for logback users). If you do not use SLF4J, this pattern is NOT automatically applied. | | |

**50.1 Introduction to Brave**

|  |  |
| --- | --- |
| [Important] | **Important** |
| Starting with version 2.0.0, Spring Cloud Sleuth uses [Brave](https://github.com/openzipkin/brave) as the tracing library. For your convenience, we embed part of the Brave’s docs here. |
| [Important] | **Important** | |
| In the vast majority of cases you need to just use the Tracer or SpanCustomizer beans from Brave that Sleuth provides. The documentation below contains a high overview of what Brave is and how it works. | |

Brave is a library used to capture and report latency information about distributed operations to Zipkin. Most users do not use Brave directly. They use libraries or frameworks rather than employ Brave on their behalf.

This module includes a tracer that creates and joins spans that model the latency of potentially distributed work. It also includes libraries to propagate the trace context over network boundaries (for example, with HTTP headers).

**50.1.1 Tracing**

Most importantly, you need a brave.Tracer, configured to [report to Zipkin](https://github.com/openzipkin/zipkin-reporter-java).

The following example setup sends trace data (spans) to Zipkin over HTTP (as opposed to Kafka):

**class** MyClass {

**private** **final** Tracer tracer;

*// Tracer will be autowired*

MyClass(Tracer tracer) {

**this**.tracer = tracer;

}

**void** doSth() {

Span span = tracer.newTrace().name("encode").start();

*// ...*

}

}

|  |  |
| --- | --- |
| [Important] | **Important** |
| If your span contains a name longer than 50 chars, then that name is truncated to 50 chars. Your names have to be explicit and concrete. Big names lead to latency issues and sometimes even thrown exceptions. |

The tracer creates and joins spans that model the latency of potentially distributed work. It can employ sampling to reduce overhead during the process, to reduce the amount of data sent to Zipkin, or both.

Spans returned by a tracer report data to Zipkin when finished or do nothing if unsampled. After starting a span, you can annotate events of interest or add tags containing details or lookup keys.

Spans have a context that includes trace identifiers that place the span at the correct spot in the tree representing the distributed operation.

**50.1.2 Local Tracing**

When tracing local code, you can run it inside a span, as shown in the following example:

*@Autowired* Tracer tracer;

Span span = tracer.newTrace().name("encode").start();

**try** {

doSomethingExpensive();

} **finally** {

span.finish();

}

In the preceding example, the span is the root of the trace. In many cases, the span is part of an existing trace. When this is the case, call newChild instead of newTrace, as shown in the following example:

*@Autowired* Tracer tracer;

Span span = tracer.newChild(root.context()).name("encode").start();

**try** {

doSomethingExpensive();

} **finally** {

span.finish();

}

**50.1.3 Customizing Spans**

Once you have a span, you can add tags to it. The tags can be used as lookup keys or details. For example, you might add a tag with your runtime version, as shown in the following example:

span.tag("clnt/finagle.version", "6.36.0");

When exposing the ability to customize spans to third parties, prefer brave.SpanCustomizer as opposed to brave.Span. The former is simpler to understand and test and does not tempt users with span lifecycle hooks.

**interface** MyTraceCallback {

**void** request(Request request, SpanCustomizer customizer);

}

Since brave.Span implements brave.SpanCustomizer, you can pass it to users, as shown in the following example:

**for** (MyTraceCallback callback : userCallbacks) {

callback.request(request, span);

}

**50.1.4 Implicitly Looking up the Current Span**

Sometimes, you do not know if a trace is in progress or not, and you do not want users to do null checks. brave.CurrentSpanCustomizer handles this problem by adding data to any span that’s in progress or drops, as shown in the following example:

Ex.

*// The user code can then inject this without a chance of it being null.*

*@Autowired* SpanCustomizer span;

**void** userCode() {

span.annotate("tx.started");

...

}

**50.1.5 RPC tracing**

|  |
| --- |
| [Tip] |
| Check for [instrumentation written here](https://github.com/openzipkin/sleuth/tree/master/instrumentation) and [Zipkin’s list](http://zipkin.io/pages/existing_instrumentations.html) before rolling your own RPC instrumentation. |

RPC tracing is often done automatically by interceptors. Behind the scenes, they add tags and events that relate to their role in an RPC operation.

The following example shows how to add a client span:

*@Autowired* Tracer tracer;

*// before you send a request, add metadata that describes the operation*

span = tracer.newTrace().name("get").type(CLIENT);

span.tag("clnt/finagle.version", "6.36.0");

span.tag(TraceKeys.HTTP\_PATH, "/api");

span.remoteEndpoint(Endpoint.builder()

.serviceName("backend")

.ipv4(127 << 24 | 1)

.port(8080).build());

*// when the request is scheduled, start the span*

span.start();

*// if you have callbacks for when data is on the wire, note those events*

span.annotate(Constants.WIRE\_SEND);

span.annotate(Constants.WIRE\_RECV);

*// when the response is complete, finish the span*

span.finish();

**One-Way tracing**

Sometimes, you need to model an asynchronous operation where there is a request but no response. In normal RPC tracing, you use span.finish() to indicate that the response was received. In one-way tracing, you use span.flush() instead, as you do not expect a response.

The following example shows how a client might model a one-way operation:

*@Autowired* Tracer tracer;

*// start a new span representing a client request*

oneWaySend = tracer.newSpan(parent).kind(Span.Kind.CLIENT);

*// Add the trace context to the request, so it can be propagated in-band*

tracing.propagation().injector(Request::addHeader)

.inject(oneWaySend.context(), request);

*// fire off the request asynchronously, totally dropping any response*

request.execute();

*// start the client side and flush instead of finish*

oneWaySend.start().flush();

The following example shows how a server might handle a one-way operation:

*@Autowired* Tracing tracing;

*@Autowired* Tracer tracer;

*// pull the context out of the incoming request*

extractor = tracing.propagation().extractor(Request::getHeader);

*// convert that context to a span which you can name and add tags to*

oneWayReceive = nextSpan(tracer, extractor.extract(request))

.name("process-request")

.kind(SERVER)

... add tags etc.

*// start the server side and flush instead of finish*

oneWayReceive.start().flush();

*// you should not modify this span anymore as it is complete. However,*

*// you can create children to represent follow-up work.*

next = tracer.newSpan(oneWayReceive.context()).name("step2").start();

**51. Sampling**

Sampling may be employed to reduce the data collected and reported out of process. When a span is not sampled, it adds no overhead (a noop).

Sampling is an up-front decision, meaning that the decision to report data is made at the first operation in a trace and that decision is propagated downstream.

By default, a global sampler applies a single rate to all traced operations. Tracer.Builder.sampler controls this setting, and it defaults to tracing every request.

**51.1 Declarative sampling**

Some applications need to sample based on the type or annotations of a java method.

Most users use a framework interceptor to automate this sort of policy. The following example shows how that might work internally:

*@Autowired* Tracing tracing;

*// derives a sample rate from an annotation on a java method*

DeclarativeSampler<Traced> sampler = DeclarativeSampler.create(Traced::sampleRate);

*@Around("@annotation(traced)")*

**public** Object traceThing(ProceedingJoinPoint pjp, Traced traced) **throws** Throwable {

Span span = tracing.tracer().newTrace(sampler.sample(traced))...

**try** {

**return** pjp.proceed();

} **finally** {

span.finish();

}

}

**51.2 Custom sampling**

Depending on what the operation is, you may want to apply different policies. For example, you might not want to trace requests to static resources such as images, or you might want to trace all requests to a new api.

Most users use a framework interceptor to automate this sort of policy. The following example shows how that might work internally:

*@Autowired* Tracer tracer;

Span newTrace(Request input) {

SamplingFlags flags = SamplingFlags.NONE;

**if** (input.url().startsWith("/experimental")) {

flags = SamplingFlags.SAMPLED;

} **else** **if** (input.url().startsWith("/static")) {

flags = SamplingFlags.NOT\_SAMPLED;

}

**return** tracer.newTrace(flags);

}

**51.3 Sampling in Spring Cloud Sleuth**

By default Spring Cloud Sleuth sets all spans to non-exportable. That means that traces appear in logs but not in any remote store. For testing the default is often enough, and it probably is all you need if you use only the logs (for example, with an ELK aggregator). If you export span data to Zipkin, there is also an Sampler.ALWAYS\_SAMPLE setting that exports everything and a ProbabilityBasedSampler setting that samples a fixed fraction of spans.

|  |
| --- |
| [Note] |
| The ProbabilityBasedSampler is the default if you use spring-cloud-sleuth-zipkin. You can configure the exports by setting spring.sleuth.sampler.probability. The passed value needs to be a double from 0.0 to 1.0. |

A sampler can be installed by creating a bean definition, as shown in the following example:

*@Bean*

**public** Sampler defaultSampler() {

**return** Sampler.ALWAYS\_SAMPLE;

}

|  |
| --- |
| [Tip] |
| You can set the HTTP header X-B3-Flags to 1, or, when doing messaging, you can set the spanFlags header to 1. Doing so forces the current span to be exportable regardless of the sampling decision. |

**52. Propagation**

Propagation is needed to ensure activities originating from the same root are collected together in the same trace. The most common propagation approach is to copy a trace context from a client by sending an RPC request to a server receiving it.

For example, when a downstream HTTP call is made, its trace context is encoded as request headers and sent along with it, as shown in the following image:

Client Span Server Span

┌──────────────────┐ ┌──────────────────┐

│ │ │ │

│ TraceContext │ Http Request Headers │ TraceContext │

│ ┌──────────────┐ │ ┌───────────────────┐ │ ┌──────────────┐ │

│ │ TraceId │ │ │ X─B3─TraceId │ │ │ TraceId │ │

│ │ │ │ │ │ │ │ │ │

│ │ ParentSpanId │ │ Extract │ X─B3─ParentSpanId │ Inject │ │ ParentSpanId │ │

│ │ ├─┼─────────>│ ├────────┼>│ │ │

│ │ SpanId │ │ │ X─B3─SpanId │ │ │ SpanId │ │

│ │ │ │ │ │ │ │ │ │

│ │ Sampled │ │ │ X─B3─Sampled │ │ │ Sampled │ │

│ └──────────────┘ │ └───────────────────┘ │ └──────────────┘ │

│ │ │ │

└──────────────────┘ └──────────────────┘

The names above are from [B3 Propagation](https://github.com/openzipkin/b3-propagation), which is built-in to Brave and has implementations in many languages and frameworks.

Most users use a framework interceptor to automate propagation. The next two examples show how that might work for a client and a server.

The following example shows how client-side propagation might work:

*@Autowired* Tracing tracing;

*// configure a function that injects a trace context into a request*

injector = tracing.propagation().injector(Request.Builder::addHeader);

*// before a request is sent, add the current span's context to it*

injector.inject(span.context(), request);

The following example shows how server-side propagation might work:

*@Autowired* Tracing tracing;

*@Autowired* Tracer tracer;

*// configure a function that extracts the trace context from a request*

extractor = tracing.propagation().extractor(Request::getHeader);

*// when a server receives a request, it joins or starts a new trace*

span = tracer.nextSpan(extractor.extract(request));

**52.1 Propagating extra fields**

Sometimes you need to propagate extra fields, such as a request ID or an alternate trace context. For example, if you are in a Cloud Foundry environment, you might want to pass the request ID, as shown in the following example:

*// when you initialize the builder, define the extra field you want to propagate*

Tracing.newBuilder().propagationFactory(

ExtraFieldPropagation.newFactory(B3Propagation.FACTORY, "x-vcap-request-id")

);

*// later, you can tag that request ID or use it in log correlation*

requestId = ExtraFieldPropagation.get("x-vcap-request-id");

You may also need to propagate a trace context that you are not using. For example, you may be in an Amazon Web Services environment but not be reporting data to X-Ray. To ensure X-Ray can co-exist correctly, pass-through its tracing header, as shown in the following example:

tracingBuilder.propagationFactory(

ExtraFieldPropagation.newFactory(B3Propagation.FACTORY, "x-amzn-trace-id")

);

|  |
| --- |
| [Tip] |
| In Spring Cloud Sleuth all elements of the tracing builder Tracing.newBuilder() are defined as beans. So if you want to pass a custom PropagationFactory, it’s enough for you to create a bean of that type and we will set it in the Tracing bean. |

**52.1.1 Prefixed fields**

If they follow a common pattern, you can also prefix fields. The following example shows how to propagate x-vcap-request-id the field as-is but send the country-code and user-id fields on the wire as x-baggage-country-code and x-baggage-user-id, respectively:

Tracing.newBuilder().propagationFactory(

ExtraFieldPropagation.newFactoryBuilder(B3Propagation.FACTORY)

.addField("x-vcap-request-id")

.addPrefixedFields("baggage-", Arrays.asList("country-code", "user-id"))

.build()

);

Later, you can call the following code to affect the country code of the current trace context:

ExtraFieldPropagation.set("country-code", "FO");

String countryCode = ExtraFieldPropagation.get("country-code");

Alternatively, if you have a reference to a trace context, you can use it explicitly, as shown in the following example:

ExtraFieldPropagation.set(span.context(), "country-code", "FO");

String countryCode = ExtraFieldPropagation.get(span.context(), "country-code");

|  |  |
| --- | --- |
| [Important] | **Important** |
| A difference from previous versions of Sleuth is that, with Brave, you must pass the list of baggage keys. There are two properties to achieve this. With the spring.sleuth.baggage-keys, you set keys that get prefixed with baggage- for HTTP calls and baggage\_ for messaging. You can also use the spring.sleuth.propagation-keys property to pass a list of prefixed keys that are whitelisted without any prefix. |

**52.1.2 Extracting a Propagated Context**

The TraceContext.Extractor<C> reads trace identifiers and sampling status from an incoming request or message. The carrier is usually a request object or headers.

This utility is used in standard instrumentation (such as HttpServerHandler`) but can also be used for custom RPC or messaging code.

TraceContextOrSamplingFlags is usually used only with Tracer.nextSpan(extracted), unless you are sharing span IDs between a client and a server.

**52.1.3 Sharing span IDs between Client and Server**

A normal instrumentation pattern is to create a span representing the server side of an RPC. Extractor.extract might return a complete trace context when applied to an incoming client request. Tracer.joinSpan attempts to continue this trace, using the same span ID if supported or creating a child span if not. When the span ID is shared, the reported data includes a flag saying so.

The following image shows an example of B3 propagation:

┌───────────────────┐ ┌───────────────────┐

Incoming Headers │ TraceContext │ │ TraceContext │

┌───────────────────┐(extract)│ ┌───────────────┐ │(join)│ ┌───────────────┐ │

│ X─B3-TraceId │─────────┼─┼> TraceId │ │──────┼─┼> TraceId │ │

│ │ │ │ │ │ │ │ │ │

│ X─B3-ParentSpanId │─────────┼─┼> ParentSpanId │ │──────┼─┼> ParentSpanId │ │

│ │ │ │ │ │ │ │ │ │

│ X─B3-SpanId │─────────┼─┼> SpanId │ │──────┼─┼> SpanId │ │

└───────────────────┘ │ │ │ │ │ │ │ │

│ │ │ │ │ │ Shared: true │ │

│ └───────────────┘ │ │ └───────────────┘ │

└───────────────────┘ └───────────────────┘

Some propagation systems forward only the parent span ID, detected when Propagation.Factory.supportsJoin() == false. In this case, a new span ID is always provisioned, and the incoming context determines the parent ID.

The following image shows an example of AWS propagation:

┌───────────────────┐ ┌───────────────────┐

x-amzn-trace-id │ TraceContext │ │ TraceContext │

┌───────────────────┐(extract)│ ┌───────────────┐ │(join)│ ┌───────────────┐ │

│ Root │─────────┼─┼> TraceId │ │──────┼─┼> TraceId │ │

│ │ │ │ │ │ │ │ │ │

│ Parent │─────────┼─┼> SpanId │ │──────┼─┼> ParentSpanId │ │

└───────────────────┘ │ └───────────────┘ │ │ │ │ │

└───────────────────┘ │ │ SpanId: New │ │

│ └───────────────┘ │

└───────────────────┘

Note: Some span reporters do not support sharing span IDs. For example, if you set Tracing.Builder.spanReporter(amazonXrayOrGoogleStackdrive), you should disable join by setting Tracing.Builder.supportsJoin(false). Doing so forces a new child span on Tracer.joinSpan().

**52.1.4 Implementing Propagation**

TraceContext.Extractor<C> is implemented by a Propagation.Factory plugin. Internally, this code creates the union type, TraceContextOrSamplingFlags, with one of the following: \* TraceContext if trace and span IDs were present. \* TraceIdContext if a trace ID was present but span IDs were not present. \* SamplingFlags if no identifiers were present.

Some Propagation implementations carry extra data from the point of extraction (for example, reading incoming headers) to injection (for example, writing outgoing headers). For example, it might carry a request ID. When implementations have extra data, they handle it as follows: \* If a TraceContext were extracted, add the extra data as TraceContext.extra(). \* Otherwise, add it as TraceContextOrSamplingFlags.extra(), which Tracer.nextSpan handles.

**53. Current Tracing Component**

Brave supports a “current tracing component” concept, which should only be used when you have no other way to get a reference. This was made for JDBC connections, as they often initialize prior to the tracing component.

The most recent tracing component instantiated is available through Tracing.current(). You can also use Tracing.currentTracer() to get only the tracer. If you use either of these methods, do not cache the result. Instead, look them up each time you need them.

**54. Current Span**

Brave supports a “current span” concept which represents the in-flight operation. You can use Tracer.currentSpan() to add custom tags to a span and Tracer.nextSpan() to create a child of whatever is in-flight.

|  |  |
| --- | --- |
| [Important] | **Important** |
| In Sleuth, you can autowire the Tracer bean to retrieve the current span via tracer.currentSpan() method. To retrieve the current context just calltracer.currentSpan().context(). To get the current trace id as String you can use the traceIdString() method like this: tracer.currentSpan().context().traceIdString(). |

**54.1 Setting a span in scope manually**

When writing new instrumentation, it is important to place a span you created in scope as the current span. Not only does doing so let users access it with Tracer.currentSpan(), but it also allows customizations such as SLF4J MDC to see the current trace IDs.

Tracer.withSpanInScope(Span) facilitates this and is most conveniently employed by using the try-with-resources idiom. Whenever external code might be invoked (such as proceeding an interceptor or otherwise), place the span in scope, as shown in the following example:

*@Autowired* Tracer tracer;

**try** (SpanInScope ws = tracer.withSpanInScope(span)) {

**return** inboundRequest.invoke();

} **finally** { *// note the scope is independent of the span*

span.finish();

}

In edge cases, you may need to clear the current span temporarily (for example, launching a task that should not be associated with the current request). To do tso, pass null to withSpanInScope, as shown in the following example:

*@Autowired* Tracer tracer;

**try** (SpanInScope cleared = tracer.withSpanInScope(null)) {

startBackgroundThread();

}

**55. Instrumentation**

Spring Cloud Sleuth automatically instruments all your Spring applications, so you should not have to do anything to activate it. The instrumentation is added by using a variety of technologies according to the stack that is available. For example, for a servlet web application, we use a Filter, and, for Spring Integration, we use ChannelInterceptors.

You can customize the keys used in span tags. To limit the volume of span data, an HTTP request is, by default, tagged only with a handful of metadata, such as the status code, the host, and the URL. You can add request headers by configuring spring.sleuth.keys.http.headers (a list of header names).

|  |
| --- |
| [Note] |
| Tags are collected and exported only if there is a Sampler that allows it. By default, there is no such Sampler, to ensure that there is no danger of accidentally collecting too much data without configuring something). |

**56. Span lifecycle**

You can do the following operations on the Span by means of brave.Tracer:

* [start](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#creating-and-finishing-spans): When you start a span, its name is assigned and the start timestamp is recorded.
* [close](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#creating-and-finishing-spans): The span gets finished (the end time of the span is recorded) and, if the span is sampled, it is eligible for collection (for example, to Zipkin).
* [continue](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#continuing-spans): A new instance of span is created. It is a copy of the one that it continues.
* [detach](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#continuing-spans): The span does not get stopped or closed. It only gets removed from the current thread.
* [create with explicit parent](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#creating-spans-with-explicit-parent): You can create a new span and set an explicit parent for it.

|  |
| --- |
| [Tip] |
| Spring Cloud Sleuth creates an instance of Tracer for you. In order to use it, you can autowire it. |

**56.1 Creating and finishing spans**

You can manually create spans by using the Tracer, as shown in the following example:

*// Start a span. If there was a span present in this thread it will become*

*// the `newSpan`'s parent.*

Span newSpan = **this**.tracer.nextSpan().name("calculateTax");

**try** (Tracer.SpanInScope ws = **this**.tracer.withSpanInScope(newSpan.start())) {

*// ...*

*// You can tag a span*

newSpan.tag("taxValue", taxValue);

*// ...*

*// You can log an event on a span*

newSpan.annotate("taxCalculated");

} **finally** {

*// Once done remember to finish the span. This will allow collecting*

*// the span to send it to Zipkin*

newSpan.finish();

}

In the preceding example, we could see how to create a new instance of the span. If there is already a span in this thread, it becomes the parent of the new span.

|  |  |
| --- | --- |
| [Important] | **Important** |
| Always clean after you create a span. Also, always finish any span that you want to send to Zipkin. |
| [Important] | **Important** | |
| If your span contains a name greater than 50 chars, that name is truncated to 50 chars. Your names have to be explicit and concrete. Big names lead to latency issues and sometimes even exceptions. | |

**56.2 Continuing Spans**

Sometimes, you do not want to create a new span but you want to continue one. An example of such a situation might be as follows:

* **AOP**: If there was already a span created before an aspect was reached, you might not want to create a new span.
* **Hystrix**: Executing a Hystrix command is most likely a logical part of the current processing. It is in fact merely a technical implementation detail that you would not necessarily want to reflect in tracing as a separate being.

To continue a span, you can use brave.Tracer, as shown in the following example:

*// let's assume that we're in a thread Y and we've received*

*// the `initialSpan` from thread X*

Span continuedSpan = **this**.tracer.toSpan(newSpan.context());

**try** {

*// ...*

*// You can tag a span*

continuedSpan.tag("taxValue", taxValue);

*// ...*

*// You can log an event on a span*

continuedSpan.annotate("taxCalculated");

} **finally** {

*// Once done remember to flush the span. That means that*

*// it will get reported but the span itself is not yet finished*

continuedSpan.flush();

}

**56.3 Creating a Span with an explicit Parent**

You might want to start a new span and provide an explicit parent of that span. Assume that the parent of a span is in one thread and you want to start a new span in another thread. In Brave, whenever you call nextSpan(), it creates a span in reference to the span that is currently in scope. You can put the span in scope and then call nextSpan(), as shown in the following example:

*// let's assume that we're in a thread Y and we've received*

*// the `initialSpan` from thread X. `initialSpan` will be the parent*

*// of the `newSpan`*

Span newSpan = null;

**try** (Tracer.SpanInScope ws = **this**.tracer.withSpanInScope(initialSpan)) {

newSpan = **this**.tracer.nextSpan().name("calculateCommission");

*// ...*

*// You can tag a span*

newSpan.tag("commissionValue", commissionValue);

*// ...*

*// You can log an event on a span*

newSpan.annotate("commissionCalculated");

} **finally** {

*// Once done remember to finish the span. This will allow collecting*

*// the span to send it to Zipkin. The tags and events set on the*

*// newSpan will not be present on the parent*

**if** (newSpan != null) {

newSpan.finish();

}

}

|  |  |
| --- | --- |
| [Important] | **Important** |
| After creating such a span, you must finish it. Otherwise it is not reported (for example, to Zipkin). |

**57. Naming spans**

Picking a span name is not a trivial task. A span name should depict an operation name. The name should be low cardinality, so it should not include identifiers.

Since there is a lot of instrumentation going on, some span names are artificial:

* controller-method-name when received by a Controller with a method name of controllerMethodName
* async for asynchronous operations done with wrapped Callable and Runnable interfaces.
* Methods annotated with @Scheduled return the simple name of the class.

Fortunately, for asynchronous processing, you can provide explicit naming.

**57.1 @SpanName Annotation**

You can name the span explicitly by using the @SpanName annotation, as shown in the following example:

*@SpanName("calculateTax")*

**class** TaxCountingRunnable **implements** Runnable {

*@Override* **public** **void** run() {

*// perform logic*

}

}

In this case, when processed in the following manner, the span is named calculateTax:

Runnable runnable = **new** TraceRunnable(tracing, spanNamer,

**new** TaxCountingRunnable());

Future<?> future = executorService.submit(runnable);

*// ... some additional logic ...*

future.get();

**57.2 toString() method**

It is pretty rare to create separate classes for Runnable or Callable. Typically, one creates an anonymous instance of those classes. You cannot annotate such classes. To overcome that limitation, if there is no @SpanName annotation present, we check whether the class has a custom implementation of the toString()method.

Running such code leads to creating a span named calculateTax, as shown in the following example:

Runnable runnable = **new** TraceRunnable(tracing, spanNamer, **new** Runnable() {

*@Override* **public** **void** run() {

*// perform logic*

}

*@Override* **public** String toString() {

**return** "calculateTax";

}

});

Future<?> future = executorService.submit(runnable);

*// ... some additional logic ...*

future.get();

**58. Managing Spans with Annotations**

You can manage spans with a variety of annotations.

**58.1 Rationale**

There are a number of good reasons to manage spans with annotations, including:

* API-agnostic means to collaborate with a span. Use of annotations lets users add to a span with no library dependency on a span api. Doing so lets Sleuth change its core API to create less impact to user code.
* Reduced surface area for basic span operations. Without this feature, you must use the span api, which has lifecycle commands that could be used incorrectly. By only exposing scope, tag, and log functionality, you can collaborate without accidentally breaking span lifecycle.
* Collaboration with runtime generated code. With libraries such as Spring Data and Feign, the implementations of interfaces are generated at runtime. Consequently, span wrapping of objects was tedious. Now you can provide annotations over interfaces and the arguments of those interfaces.

**58.2 Creating New Spans**

If you do not want to create local spans manually, you can use the @NewSpan annotation. Also, we provide the @SpanTag annotation to add tags in an automated fashion.

Now we can consider some examples of usage.

*@NewSpan*

**void** testMethod();

Annotating the method without any parameter leads to creating a new span whose name equals the annotated method name.

*@NewSpan("customNameOnTestMethod4")*

**void** testMethod4();

If you provide the value in the annotation (either directly or by setting the name parameter), the created span has the provided value as the name.

*// method declaration*

*@NewSpan(name = "customNameOnTestMethod5")*

**void** testMethod5(*@SpanTag("testTag")* String param);

*// and method execution*

**this**.testBean.testMethod5("test");

You can combine both the name and a tag. Let’s focus on the latter. In this case, the value of the annotated method’s parameter runtime value becomes the value of the tag. In our sample, the tag key is testTag, and the tag value is test.

*@NewSpan(name = "customNameOnTestMethod3")*

*@Override*

**public** **void** testMethod3() {

}

You can place the @NewSpan annotation on both the class and an interface. If you override the interface’s method and provide a different value for the @NewSpanannotation, the most concrete one wins (in this case customNameOnTestMethod3 is set).

**58.3 Continuing Spans**

If you want to add tags and annotations to an existing span, you can use the @ContinueSpan annotation, as shown in the following example:

*// method declaration*

*@ContinueSpan(log = "testMethod11")*

**void** testMethod11(*@SpanTag("testTag11")* String param);

*// method execution*

**this**.testBean.testMethod11("test");

**this**.testBean.testMethod13();

(Note that, in contrast with the @NewSpan annotation ,you can also add logs with the log parameter.)

That way, the span gets continued and:

* Log entries named testMethod11.before and testMethod11.after are created.
* If an exception is thrown, a log entry named testMethod11.afterFailure is also created.
* A tag with a key of testTag11 and a value of test is created.

**58.4 Advanced Tag Setting**

There are 3 different ways to add tags to a span. All of them are controlled by the SpanTag annotation. The precedence is as follows:

1. Try with a bean of TagValueResolver type and a provided name.
2. If the bean name has not been provided, try to evaluate an expression. We search for a TagValueExpressionResolver bean. The default implementation uses SPEL expression resolution. **IMPORTANT** You can only reference properties from the SPEL expression. Method execution is not allowed due to security constraints.
3. If we do not find any expression to evaluate, return the toString() value of the parameter.

**58.4.1 Custom extractor**

The value of the tag for the following method is computed by an implementation of TagValueResolver interface. Its class name has to be passed as the value of the resolver attribute.

Consider the following annotated method:

*@NewSpan*

**public** **void** getAnnotationForTagValueResolver(*@SpanTag(key = "test", resolver = TagValueResolver.class)* String test) {

}

Now further consider the following TagValueResolver bean implementation:

*@Bean(name = "myCustomTagValueResolver")*

**public** TagValueResolver tagValueResolver() {

**return** parameter -> "Value from myCustomTagValueResolver";

}

The two preceding examples lead to setting a tag value equal to Value from myCustomTagValueResolver.

**58.4.2 Resolving Expressions for a Value**

Consider the following annotated method:

*@NewSpan*

**public** **void** getAnnotationForTagValueExpression(*@SpanTag(key = "test", expression = "'hello' + ' characters'")* String test) {

}

No custom implementation of a TagValueExpressionResolver leads to evaluation of the SPEL expression, and a tag with a value of 4 characters is set on the span. If you want to use some other expression resolution mechanism, you can create your own implementation of the bean.

**58.4.3 Using the toString() method**

Consider the following annotated method:

*@NewSpan*

**public** **void** getAnnotationForArgumentToString(*@SpanTag("test")* Long param) {

}

Running the preceding method with a value of 15 leads to setting a tag with a String value of "15".

**59. Customizations**

**59.1 HTTP**

If a customization of client / server parsing of the HTTP related spans is required, just register a bean of type brave.http.HttpClientParser orbrave.http.HttpServerParser. If client /server sampling is required, just register a bean of type brave.http.HttpSampler and name the beansleuthClientSampler for client sampler and sleuthServerSampler for server sampler. For your convenience the @ClientSampler and @ServerSamplerannotations can be used to inject the proper beans or to reference the bean names via their static String NAME fields.

Check out Brave’s code to see an example of how to make a path-based sampler <https://github.com/openzipkin/brave/tree/master/instrumentation/http#sampling-policy>

If you want to completely rewrite the HttpTracing bean you can use the SkipPatternProvider interface to retrieve the URL Pattern for spans that should be not sampled. Below you can see an example of usage of SkipPatternProvider inside a server side, HttpSampler.

*@Configuration*

**class** Config {

*@Bean(name = ServerSampler.NAME)*

HttpSampler myHttpSampler(SkipPatternProvider provider) {

Pattern pattern = provider.skipPattern();

**return** **new** HttpSampler() {

*@Override* **public** <Req> Boolean trySample(HttpAdapter<Req, ?> adapter, Req request) {

String url = adapter.path(request);

**boolean** shouldSkip = pattern.matcher(url).matches();

**if** (shouldSkip) {

**return** false;

}

**return** null;

}

};

}

}

**59.2 TracingFilter**

You can also modify the behavior of the TracingFilter, which is the component that is responsible for processing the input HTTP request and adding tags basing on the HTTP response. You can customize the tags or modify the response headers by registering your own instance of the TracingFilter bean.

In the following example, we register the TracingFilter bean, add the ZIPKIN-TRACE-ID response header containing the current Span’s trace id, and add a tag with key custom and a value tag to the span.

*@Component*

*@Order(TraceWebServletAutoConfiguration.TRACING\_FILTER\_ORDER + 1)*

**class** MyFilter **extends** GenericFilterBean {

**private** **final** Tracer tracer;

MyFilter(Tracer tracer) {

**this**.tracer = tracer;

}

*@Override* **public** **void** doFilter(ServletRequest request, ServletResponse response,

FilterChain chain) **throws** IOException, ServletException {

Span currentSpan = **this**.tracer.currentSpan();

**if** (currentSpan == null) {

chain.doFilter(request, response);

**return**;

}

*// for readability we're returning trace id in a hex form*

((HttpServletResponse) response)

.addHeader("ZIPKIN-TRACE-ID",

currentSpan.context().traceIdString());

*// we can also add some custom tags*

currentSpan.tag("custom", "tag");

chain.doFilter(request, response);

}

}

**59.3 Custom service name**

By default, Sleuth assumes that, when you send a span to Zipkin, you want the span’s service name to be equal to the value of the spring.application.nameproperty. That is not always the case, though. There are situations in which you want to explicitly provide a different service name for all spans coming from your application. To achieve that, you can pass the following property to your application to override that value (the example is for a service named myService):

spring.zipkin.service.name: myService

**59.4 Customization of Reported Spans**

Before reporting spans (for example, to Zipkin) you may want to modify that span in some way. You can do so by using the SpanAdjuster interface.

In Sleuth, we generate spans with a fixed name. Some users want to modify the name depending on values of tags. You can implement the SpanAdjuster interface to alter that name.

The following example shows how to register two beans that implement SpanAdjuster:

*@Bean* SpanAdjuster adjusterOne() {

**return** span -> span.toBuilder().name("foo").build();

}

*@Bean* SpanAdjuster adjusterTwo() {

**return** span -> span.toBuilder().name(span.name() + " bar").build();

}

The preceding example results in changing the name of the reported span to foo bar, just before it gets reported (for example, to Zipkin).

**59.5 Host Locator**

|  |  |
| --- | --- |
| [Important] | **Important** |
| This section is about defining **host** from service discovery. It is **NOT** about finding Zipkin through service discovery. |

To define the host that corresponds to a particular span, we need to resolve the host name and port. The default approach is to take these values from server properties. If those are not set, we try to retrieve the host name from the network interfaces.

If you have the discovery client enabled and prefer to retrieve the host address from the registered instance in a service registry, you have to set the spring.zipkin.locator.discovery.enabled property (it is applicable for both HTTP-based and Stream-based span reporting), as follows:

spring.zipkin.locator.discovery.enabled: **true**

**60. Sending Spans to Zipkin**

By default, if you add spring-cloud-starter-zipkin as a dependency to your project, when the span is closed, it is sent to Zipkin over HTTP. The communication is asynchronous. You can configure the URL by setting the spring.zipkin.baseUrl property, as follows:

spring.zipkin.baseUrl: http://192.168.99.100:9411/

If you want to find Zipkin through service discovery, you can pass the Zipkin’s service ID inside the URL, as shown in the following example for zipkinserver service ID:

spring.zipkin.baseUrl: http://zipkinserver/

To disable this feature just set spring.zipkin.discoveryClientEnabled to `false.

When the Discovery Client feature is enabled, Sleuth uses LoadBalancerClient to find the URL of the Zipkin Server. It means that you can set up the load balancing configuration e.g. via Ribbon.

zipkinserver:

ribbon:

ListOfServers: host1,host2

If you have web, rabbit, or kafka together on the classpath, you might need to pick the means by which you would like to send spans to zipkin. To do so, set web, rabbit, or kafka to the spring.zipkin.sender.type property. The following example shows setting the sender type for web:

spring.zipkin.sender.type: web

To customize the RestTemplate that sends spans to Zipkin via HTTP, you can register the ZipkinRestTemplateCustomizer bean.

*@Configuration*

**class** MyConfig {

*@Bean* ZipkinRestTemplateCustomizer myCustomizer() {

**return** **new** ZipkinRestTemplateCustomizer() {

*@Override*

**void** customize(RestTemplate restTemplate) {

*// customize the RestTemplate*

}

};

}

}

If, however, you would like to control the full process of creating the RestTemplate object, you will have to create a bean of zipkin2.reporter.Sender type.

*@Bean* Sender myRestTemplateSender(ZipkinProperties zipkin,

ZipkinRestTemplateCustomizer zipkinRestTemplateCustomizer) {

RestTemplate restTemplate = mySuperCustomRestTemplate();

zipkinRestTemplateCustomizer.customize(restTemplate);

**return** myCustomSender(zipkin, restTemplate);

}

**61. Zipkin Stream Span Consumer**

|  |  |
| --- | --- |
| [Important] | **Important** |
| We recommend using Zipkin’s native support for message-based span sending. Starting from the Edgware release, the Zipkin Stream server is deprecated. In the Finchley release, it got removed. |

If for some reason you need to create the deprecated Stream Zipkin server, see the [Dalston Documentation](https://cloud.spring.io/spring-cloud-static/Dalston.SR4/multi/multi__span_data_as_messages.html#_zipkin_consumer).

**62. Integrations**

**62.1 OpenTracing**

Spring Cloud Sleuth is compatible with [OpenTracing](http://opentracing.io/). If you have OpenTracing on the classpath, we automatically register the OpenTracing Tracer bean. If you wish to disable this, set spring.sleuth.opentracing.enabled to false

**62.2 Runnable and Callable**

If you wrap your logic in Runnable or Callable, you can wrap those classes in their Sleuth representative, as shown in the following example for Runnable:

Runnable runnable = **new** Runnable() {

*@Override*

**public** **void** run() {

*// do some work*

}

*@Override*

**public** String toString() {

**return** "spanNameFromToStringMethod";

}

};

*// Manual `TraceRunnable` creation with explicit "calculateTax" Span name*

Runnable traceRunnable = **new** TraceRunnable(tracing, spanNamer, runnable,

"calculateTax");

*// Wrapping `Runnable` with `Tracing`. That way the current span will be available*

*// in the thread of `Runnable`*

Runnable traceRunnableFromTracer = tracing.currentTraceContext().wrap(runnable);

The following example shows how to do so for Callable:

Callable<String> callable = **new** Callable<String>() {

*@Override*

**public** String call() **throws** Exception {

**return** someLogic();

}

*@Override*

**public** String toString() {

**return** "spanNameFromToStringMethod";

}

};

*// Manual `TraceCallable` creation with explicit "calculateTax" Span name*

Callable<String> traceCallable = **new** TraceCallable<>(tracing, spanNamer, callable,

"calculateTax");

*// Wrapping `Callable` with `Tracing`. That way the current span will be available*

*// in the thread of `Callable`*

Callable<String> traceCallableFromTracer = tracing.currentTraceContext().wrap(callable);

That way, you ensure that a new span is created and closed for each execution.

**62.3 Hystrix**

**62.3.1 Custom Concurrency Strategy**

We register a custom [HystrixConcurrencyStrategy](https://github.com/Netflix/Hystrix/wiki/Plugins#concurrencystrategy) called TraceCallable that wraps all Callable instances in their Sleuth representative. The strategy either starts or continues a span, depending on whether tracing was already going on before the Hystrix command was called. To disable the custom Hystrix Concurrency Strategy, set the spring.sleuth.hystrix.strategy.enabled to false.

**62.3.2 Manual Command setting**

Assume that you have the following HystrixCommand:

HystrixCommand<String> hystrixCommand = **new** HystrixCommand<String>(setter) {

*@Override*

**protected** String run() **throws** Exception {

**return** someLogic();

}

};

To pass the tracing information, you have to wrap the same logic in the Sleuth version of the HystrixCommand, which is called TraceCommand, as shown in the following example:

TraceCommand<String> traceCommand = **new** TraceCommand<String>(tracer, setter) {

*@Override*

**public** String doRun() **throws** Exception {

**return** someLogic();

}

};

**62.4 RxJava**

We registering a custom [RxJavaSchedulersHook](https://github.com/ReactiveX/RxJava/wiki/Plugins#rxjavaschedulershook) that wraps all Action0 instances in their Sleuth representative, which is called TraceAction. The hook either starts or continues a span, depending on whether tracing was already going on before the Action was scheduled. To disable the custom RxJavaSchedulersHook, set the spring.sleuth.rxjava.schedulers.hook.enabled to false.

You can define a list of regular expressions for thread names for which you do not want spans to be created. To do so, provide a comma-separated list of regular expressions in the spring.sleuth.rxjava.schedulers.ignoredthreads property.

|  |  |
| --- | --- |
| [Important] | **Important** |
| The suggest approach to reactive programming and Sleuth is to use the Reactor support. |

**62.5 HTTP integration**

Features from this section can be disabled by setting the spring.sleuth.web.enabled property with value equal to false.

**62.5.1 HTTP Filter**

Through the TracingFilter, all sampled incoming requests result in creation of a Span. That Span’s name is http: + the path to which the request was sent. For example, if the request was sent to /this/that then the name will be http:/this/that. You can configure which URIs you would like to skip by setting the spring.sleuth.web.skipPattern property. If you have ManagementServerProperties on classpath, its value of contextPath gets appended to the provided skip pattern. If you want to reuse the Sleuth’s default skip patterns and just append your own, pass those patterns by using the spring.sleuth.web.additionalSkipPattern.

**62.5.2 HandlerInterceptor**

Since we want the span names to be precise, we use a TraceHandlerInterceptor that either wraps an existing HandlerInterceptor or is added directly to the list of existing HandlerInterceptors. The TraceHandlerInterceptor adds a special request attribute to the given HttpServletRequest. If the the TracingFilterdoes not see this attribute, it creates a “fallback” span, which is an additional span created on the server side so that the trace is presented properly in the UI. If that happens, there is probably missing instrumentation. In that case, please file an issue in Spring Cloud Sleuth.

**62.5.3 Async Servlet support**

If your controller returns a Callable or a WebAsyncTask, Spring Cloud Sleuth continues the existing span instead of creating a new one.

**62.5.4 WebFlux support**

Through TraceWebFilter, all sampled incoming requests result in creation of a Span. That Span’s name is http: + the path to which the request was sent. For example, if the request was sent to /this/that, the name is http:/this/that. You can configure which URIs you would like to skip by using the spring.sleuth.web.skipPattern property. If you have ManagementServerProperties on the classpath, its value of contextPath gets appended to the provided skip pattern. If you want to reuse Sleuth’s default skip patterns and append your own, pass those patterns by using the spring.sleuth.web.additionalSkipPattern.

**62.5.5 Dubbo RPC support**

Via the integration with Brave, Spring Cloud Sleuth supports [Dubbo](http://dubbo.io/). It’s enough to add the brave-instrumentation-dubbo-rpc dependency:

<dependency>

<groupId>io.zipkin.brave</groupId>

<artifactId>brave-instrumentation-dubbo-rpc</artifactId>

</dependency>

You need to also set a dubbo.properties file with the following contents:

dubbo.provider.filter=tracing

dubbo.consumer.filter=tracing

You can read more about Brave - Dubbo integration [here](https://github.com/openzipkin/brave/tree/master/instrumentation/dubbo-rpc). An example of Spring Cloud Sleuth and Dubbo can be found [here](https://github.com/openzipkin/sleuth-webmvc-example/compare/add-dubbo-tracing).

**62.6 HTTP Client Integration**

**62.6.1 Synchronous Rest Template**

We inject a RestTemplate interceptor to ensure that all the tracing information is passed to the requests. Each time a call is made, a new Span is created. It gets closed upon receiving the response. To block the synchronous RestTemplate features, set spring.sleuth.web.client.enabled to false.

|  |  |
| --- | --- |
| [Important] | **Important** |
| You have to register RestTemplate as a bean so that the interceptors get injected. If you create a RestTemplate instance with a new keyword, the instrumentation does NOT work. |

**62.6.2 Asynchronous Rest Template**

|  |  |
| --- | --- |
| [Important] | **Important** |
| Starting with Sleuth 2.0.0, we no longer register a bean of AsyncRestTemplate type. It is up to you to create such a bean. Then we instrument it. |

To block the AsyncRestTemplate features, set spring.sleuth.web.async.client.enabled to false. To disable creation of the default TraceAsyncClientHttpRequestFactoryWrapper, set spring.sleuth.web.async.client.factory.enabled to false. If you do not want to create AsyncRestClient at all, set spring.sleuth.web.async.client.template.enabled to false.

**Multiple Asynchronous Rest Templates**

Sometimes you need to use multiple implementations of the Asynchronous Rest Template. In the following snippet, you can see an example of how to set up such a custom AsyncRestTemplate:

*@Configuration*

*@EnableAutoConfiguration*

**static** **class** Config {

*@Bean(name = "customAsyncRestTemplate")*

**public** AsyncRestTemplate traceAsyncRestTemplate() {

**return** **new** AsyncRestTemplate(asyncClientFactory(), clientHttpRequestFactory());

}

**private** ClientHttpRequestFactory clientHttpRequestFactory() {

ClientHttpRequestFactory clientHttpRequestFactory = **new** CustomClientHttpRequestFactory();

*//CUSTOMIZE HERE*

**return** clientHttpRequestFactory;

}

**private** AsyncClientHttpRequestFactory asyncClientFactory() {

AsyncClientHttpRequestFactory factory = **new** CustomAsyncClientHttpRequestFactory();

*//CUSTOMIZE HERE*

**return** factory;

}

}

**62.6.3 WebClient**

We inject a ExchangeFilterFunction implementation that creates a span and, through on-success and on-error callbacks, takes care of closing client-side spans.

To block this feature, set spring.sleuth.web.client.enabled to false.

|  |  |
| --- | --- |
| [Important] | **Important** |
| You have to register WebClient as a bean so that the tracing instrumentation gets applied. If you create a WebClient instance with a new keyword, the instrumentation does NOT work. |

**62.6.4 Traverson**

If you use the [Traverson](https://docs.spring.io/spring-hateoas/docs/current/reference/html/#client.traverson) library, you can inject a RestTemplate as a bean into your Traverson object. Since RestTemplate is already intercepted, you get full support for tracing in your client. The following pseudo code shows how to do that:

*@Autowired* RestTemplate restTemplate;

Traverson traverson = **new** Traverson(URI.create("http://some/address"),

MediaType.APPLICATION\_JSON, MediaType.APPLICATION\_JSON\_UTF8).setRestOperations(restTemplate);

*// use Traverson*

**62.6.5 Apache HttpClientBuilder and HttpAsyncClientBuilder**

We instrument the HttpClientBuilder and HttpAsyncClientBuilder so that tracing context gets injected to the sent requests.

To block these features, set spring.sleuth.web.client.enabled to false.

**62.6.6 Netty HttpClient**

We instrument the Netty’s HttpClient.

To block this feature, set spring.sleuth.web.client.enabled to false.

|  |  |
| --- | --- |
| [Important] | **Important** |
| You have to register HttpClient as a bean so that the instrumentation happens. If you create a HttpClient instance with a new keyword, the instrumentation does NOT work. |

**62.6.7 UserInfoRestTemplateCustomizer**

We instrument the Spring Security’s UserInfoRestTemplateCustomizer.

To block this feature, set spring.sleuth.web.client.enabled to false.

**62.7 Feign**

By default, Spring Cloud Sleuth provides integration with Feign through TraceFeignClientAutoConfiguration. You can disable it entirely by setting spring.sleuth.feign.enabled to false. If you do so, no Feign-related instrumentation take place.

Part of Feign instrumentation is done through a FeignBeanPostProcessor. You can disable it by setting spring.sleuth.feign.processor.enabled to false. If you set it to false, Spring Cloud Sleuth does not instrument any of your custom Feign components. However, all the default instrumentation is still there.

**62.8 Asynchronous Communication**

**62.8.1 @Async Annotated methods**

In Spring Cloud Sleuth, we instrument async-related components so that the tracing information is passed between threads. You can disable this behavior by setting the value of spring.sleuth.async.enabled to false.

If you annotate your method with @Async, we automatically create a new Span with the following characteristics:

* If the method is annotated with @SpanName, the value of the annotation is the Span’s name.
* If the method is not annotated with @SpanName, the Span name is the annotated method name.
* The span is tagged with the method’s class name and method name.

**62.8.2 @Scheduled Annotated Methods**

In Spring Cloud Sleuth, we instrument scheduled method execution so that the tracing information is passed between threads. You can disable this behavior by setting the value of spring.sleuth.scheduled.enabled to false.

If you annotate your method with @Scheduled, we automatically create a new span with the following characteristics:

* The span name is the annotated method name.
* The span is tagged with the method’s class name and method name.

If you want to skip span creation for some @Scheduled annotated classes, you can set the spring.sleuth.scheduled.skipPattern with a regular expression that matches the fully qualified name of the @Scheduled annotated class. If you use spring-cloud-sleuth-stream and spring-cloud-netflix-hystrix-streamtogether, a span is created for each Hystrix metrics and sent to Zipkin. This behavior may be annoying. That’s why, by default, spring.sleuth.scheduled.skipPattern=org.springframework.cloud.netflix.hystrix.stream.HystrixStreamTask.

**62.8.3 Executor, ExecutorService, and ScheduledExecutorService**

We provide LazyTraceExecutor, TraceableExecutorService, and TraceableScheduledExecutorService. Those implementations create spans each time a new task is submitted, invoked, or scheduled.

The following example shows how to pass tracing information with TraceableExecutorService when working with CompletableFuture:

CompletableFuture<Long> completableFuture = CompletableFuture.supplyAsync(() -> {

*// perform some logic*

**return** 1\_000\_000L;

}, **new** TraceableExecutorService(beanFactory, executorService,

*// 'calculateTax' explicitly names the span - this param is optional*

"calculateTax"));

|  |  |
| --- | --- |
| [Important] | **Important** |
| Sleuth does not work with parallelStream() out of the box. If you want to have the tracing information propagated through the stream, you have to use the approach with supplyAsync(…​), as shown earlier. |

**Customization of Executors**

Sometimes, you need to set up a custom instance of the AsyncExecutor. The following example shows how to set up such a custom Executor:

*@Configuration*

*@EnableAutoConfiguration*

*@EnableAsync*

**static** **class** CustomExecutorConfig **extends** AsyncConfigurerSupport {

*@Autowired* BeanFactory beanFactory;

*@Override* **public** Executor getAsyncExecutor() {

ThreadPoolTaskExecutor executor = **new** ThreadPoolTaskExecutor();

*// CUSTOMIZE HERE*

executor.setCorePoolSize(7);

executor.setMaxPoolSize(42);

executor.setQueueCapacity(11);

executor.setThreadNamePrefix("MyExecutor-");

*// DON'T FORGET TO INITIALIZE*

executor.initialize();

**return** **new** LazyTraceExecutor(**this**.beanFactory, executor);

}

}

**62.9 Messaging**

Features from this section can be disabled by setting the spring.sleuth.messaging.enabled property with value equal to false.

**62.9.1 Spring Integration and Spring Cloud Stream**

Spring Cloud Sleuth integrates with [Spring Integration](https://projects.spring.io/spring-integration/). It creates spans for publish and subscribe events. To disable Spring Integration instrumentation, set spring.sleuth.integration.enabled to false.

You can provide the spring.sleuth.integration.patterns pattern to explicitly provide the names of channels that you want to include for tracing. By default, all channels but hystrixStreamOutput channel are included.

|  |  |
| --- | --- |
| [Important] | **Important** |
| When using the Executor to build a Spring Integration IntegrationFlow, you must use the untraced version of the Executor. Decorating the Spring Integration Executor Channel with TraceableExecutorService causes the spans to be improperly closed. |

**62.9.2 Spring RabbitMq**

We instrument the RabbitTemplate so that tracing headers get injected into the message.

To block this feature, set spring.sleuth.messaging.rabbit.enabled to false.

**62.9.3 Spring Kafka**

We instrument the Spring Kafka’s ProducerFactory and ConsumerFactory so that tracing headers get injected into the created Spring Kafka’s Producer and Consumer.

To block this feature, set spring.sleuth.messaging.kafka.enabled to false.

|  |
| --- |
| [Note] |
| We do not support context propagation via @KafkaListener annotation. Check [this issue for more information](https://github.com/spring-cloud/spring-cloud-sleuth/issues/1001). |

**62.10 Zuul**

We instrument the Zuul Ribbon integration by enriching the Ribbon requests with tracing information. To disable Zuul support, set the spring.sleuth.zuul.enabledproperty to false.

**63. Running examples**

You can see the running examples deployed in the [Pivotal Web Services](https://run.pivotal.io/). Check them out at the following links:

* [Zipkin for apps presented in the samples to the top](https://docssleuth-zipkin-server.cfapps.io/). First make a request to [Service 1](https://docssleuth-service1.cfapps.io/start) and then check out the trace in Zipkin.
* [Zipkin for Brewery on PWS](https://docsbrewing-zipkin-server.cfapps.io/), its [Github Code](https://github.com/spring-cloud-samples/brewery). Ensure that you’ve picked the lookback period of 7 days. If there are no traces, go to [Presenting application](https://docsbrewing-presenting.cfapps.io/) and order some beers. Then check Zipkin for traces.

**Part IX. Spring Cloud Consul**

**Finchley.SR1**

This project provides Consul integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms. With a few simple annotations you can quickly enable and configure the common patterns inside your application and build large distributed systems with Consul based components. The patterns provided include Service Discovery, Control Bus and Configuration. Intelligent Routing (Zuul) and Client Side Load Balancing (Ribbon), Circuit Breaker (Hystrix) are provided by integration with Spring Cloud Netflix.

**64. Install Consul**

Please see the [installation documentation](https://www.consul.io/intro/getting-started/install.html) for instructions on how to install Consul.

**65. Consul Agent**

A Consul Agent client must be available to all Spring Cloud Consul applications. By default, the Agent client is expected to be at localhost:8500. See the [Agent documentation](https://consul.io/docs/agent/basics.html) for specifics on how to start an Agent client and how to connect to a cluster of Consul Agent Servers. For development, after you have installed consul, you may start a Consul Agent using the following command:

./src/main/bash/local\_run\_consul.sh

This will start an agent in server mode on port 8500, with the ui available at [http://localhost:8500](http://localhost:8500/)

**66. Service Discovery with Consul**

Service Discovery is one of the key tenets of a microservice based architecture. Trying to hand configure each client or some form of convention can be very difficult to do and can be very brittle. Consul provides Service Discovery services via an [HTTP API](https://www.consul.io/docs/agent/http.html) and [DNS](https://www.consul.io/docs/agent/dns.html). Spring Cloud Consul leverages the HTTP API for service registration and discovery. This does not prevent non-Spring Cloud applications from leveraging the DNS interface. Consul Agents servers are run in a [cluster](https://www.consul.io/docs/internals/architecture.html) that communicates via a [gossip protocol](https://www.consul.io/docs/internals/gossip.html) and uses the [Raft consensus protocol](https://www.consul.io/docs/internals/consensus.html).

**66.1 How to activate**

To activate Consul Service Discovery use the starter with group org.springframework.cloud and artifact id spring-cloud-starter-consul-discovery. See the [Spring Cloud Project page](https://projects.spring.io/spring-cloud/) for details on setting up your build system with the current Spring Cloud Release Train.

**66.2 Registering with Consul**

When a client registers with Consul, it provides meta-data about itself such as host and port, id, name and tags. An HTTP [Check](https://www.consul.io/docs/agent/checks.html) is created by default that Consul hits the /health endpoint every 10 seconds. If the health check fails, the service instance is marked as critical.

Example Consul client:

*@SpringBootApplication*

*@RestController*

**public** **class** Application {

*@RequestMapping("/")*

**public** String home() {

**return** "Hello world";

}

**public** **static** **void** main(String[] args) {

**new** SpringApplicationBuilder(Application.**class**).web(true).run(args);

}

}

(i.e. utterly normal Spring Boot app). If the Consul client is located somewhere other than localhost:8500, the configuration is required to locate the client. Example:

**application.yml.**

spring:

cloud:

consul:

host: localhost

port: 8500

|  |  |
| --- | --- |
| [Caution] | **Caution** |
| If you use [Spring Cloud Consul Config](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-consul-config), the above values will need to be placed in bootstrap.yml instead of application.yml. |

The default service name, instance id and port, taken from the Environment, are ${spring.application.name}, the Spring Context ID and ${server.port}respectively.

To disable the Consul Discovery Client you can set spring.cloud.consul.discovery.enabled to false.

To disable the service registration you can set spring.cloud.consul.discovery.register to false.

**66.3 HTTP Health Check**

The health check for a Consul instance defaults to "/health", which is the default locations of a useful endpoint in a Spring Boot Actuator application. You need to change these, even for an Actuator application if you use a non-default context path or servlet path (e.g. server.servletPath=/foo) or management endpoint path (e.g. management.server.servlet.context-path=/admin). The interval that Consul uses to check the health endpoint may also be configured. "10s" and "1m" represent 10 seconds and 1 minute respectively. Example:

**application.yml.**

spring:

cloud:

consul:

discovery:

healthCheckPath: ${management.server.servlet.context-path}/health

healthCheckInterval: 15s

You can disable the health check by setting management.health.consul.enabled=false.

**66.3.1 Metadata and Consul tags**

Consul does not yet support metadata on services. Spring Cloud’s ServiceInstance has a Map<String, String> metadata field. Spring Cloud Consul uses Consul tags to approximate metadata until Consul officially supports metadata. Tags with the form key=value will be split and used as a Map key and value respectively. Tags without the equal = sign, will be used as both the key and value.

**application.yml.**

spring:

cloud:

consul:

discovery:

tags: foo=bar, baz

The above configuration will result in a map with foo→bar and baz→baz.

**66.3.2 Making the Consul Instance ID Unique**

By default a consul instance is registered with an ID that is equal to its Spring Application Context ID. By default, the Spring Application Context ID is ${spring.application.name}:comma,separated,profiles:${server.port}. For most cases, this will allow multiple instances of one service to run on one machine. If further uniqueness is required, Using Spring Cloud you can override this by providing a unique identifier in spring.cloud.consul.discovery.instanceId. For example:

**application.yml.**

spring:

cloud:

consul:

discovery:

instanceId: ${spring.application.name}:${vcap.application.instance\_id:${spring.application.instance\_id:${random.value}}}

With this metadata, and multiple service instances deployed on localhost, the random value will kick in there to make the instance unique. In Cloudfoundry the vcap.application.instance\_id will be populated automatically in a Spring Boot application, so the random value will not be needed.

**66.4 Looking up services**

**66.4.1 Using Ribbon**

Spring Cloud has support for [Feign](https://github.com/spring-cloud/spring-cloud-netflix/blob/master/docs/src/main/asciidoc/spring-cloud-netflix.adoc#spring-cloud-feign) (a REST client builder) and also [Spring RestTemplate](https://github.com/spring-cloud/spring-cloud-netflix/blob/master/docs/src/main/asciidoc/spring-cloud-netflix.adoc#spring-cloud-ribbon) for looking up services using the logical service names/ids instead of physical URLs. Both Feign and the discovery-aware RestTemplate utilize [Ribbon](https://cloud.spring.io/spring-cloud-netflix/single/spring-cloud-netflix.html#spring-cloud-ribbon) for client-side load balancing.

If you want to access service STORES using the RestTemplate simply declare:

@LoadBalanced

@Bean

public RestTemplate loadbalancedRestTemplate() {

new RestTemplate();

}

and use it like this (notice how we use the STORES service name/id from Consul instead of a fully qualified domainname):

@Autowired

RestTemplate restTemplate;

public String getFirstProduct() {

return this.restTemplate.getForObject("https://STORES/products/1", String.class);

}

If you have Consul clusters in multiple datacenters and you want to access a service in another datacenter a service name/id alone is not enough. In that case you use property spring.cloud.consul.discovery.datacenters.STORES=dc-west where STORES is the service name/id and dc-west is the datacenter where the STORES service lives.

**66.4.2 Using the DiscoveryClient**

You can also use the org.springframework.cloud.client.discovery.DiscoveryClient which provides a simple API for discovery clients that is not specific to Netflix, e.g.

@Autowired

private DiscoveryClient discoveryClient;

public String serviceUrl() {

List<ServiceInstance> list = discoveryClient.getInstances("STORES");

if (list != null && list.size() > 0 ) {

return list.get(0).getUri();

}

return null;

}

**66.5 Consul Catalog Watch**

The Consul Catalog Watch takes advantage of the ability of consul to [watch services](https://www.consul.io/docs/agent/watches.html#services). The Catalog Watch makes a blocking Consul HTTP API call to determine if any services have changed. If there is new service data a Heartbeat Event is published.

To change the frequency of when the Config Watch is called change spring.cloud.consul.config.discovery.catalog-services-watch-delay. The default value is 1000, which is in milliseconds. The delay is the amount of time after the end of the previous invocation and the start of the next.

To disable the Catalog Watch set spring.cloud.consul.discovery.catalogServicesWatch.enabled=false.

The watch uses a Spring TaskScheduler to schedule the call to consul. By default it is a ThreadPoolTaskScheduler with a poolSize of 1. To change the TaskScheduler, create a bean of type TaskScheduler named with the ConsulDiscoveryClientConfiguration.CATALOG\_WATCH\_TASK\_SCHEDULER\_NAMEconstant.

**67. Distributed Configuration with Consul**

Consul provides a [Key/Value Store](https://consul.io/docs/agent/http/kv.html) for storing configuration and other metadata. Spring Cloud Consul Config is an alternative to the [Config Server and Client](https://github.com/spring-cloud/spring-cloud-config). Configuration is loaded into the Spring Environment during the special "bootstrap" phase. Configuration is stored in the /config folder by default. Multiple PropertySource instances are created based on the application’s name and the active profiles that mimicks the Spring Cloud Config order of resolving properties. For example, an application with the name "testApp" and with the "dev" profile will have the following property sources created:

config/testApp,dev/

config/testApp/

config/application,dev/

config/application/

The most specific property source is at the top, with the least specific at the bottom. Properties in the config/application folder are applicable to all applications using consul for configuration. Properties in the config/testApp folder are only available to the instances of the service named "testApp".

Configuration is currently read on startup of the application. Sending a HTTP POST to /refresh will cause the configuration to be reloaded. [Section 67.3, “Config Watch”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-consul-config-watch) will also automatically detect changes and reload the application context.

**67.1 How to activate**

To get started with Consul Configuration use the starter with group org.springframework.cloud and artifact id spring-cloud-starter-consul-config. See the [Spring Cloud Project page](https://projects.spring.io/spring-cloud/) for details on setting up your build system with the current Spring Cloud Release Train.

This will enable auto-configuration that will setup Spring Cloud Consul Config.

**67.2 Customizing**

Consul Config may be customized using the following properties:

**bootstrap.yml.**

spring:

cloud:

consul:

config:

enabled: true

prefix: configuration

defaultContext: apps

profileSeparator: '::'

* enabled setting this value to "false" disables Consul Config
* prefix sets the base folder for configuration values
* defaultContext sets the folder name used by all applications
* profileSeparator sets the value of the separator used to separate the profile name in property sources with profiles

**67.3 Config Watch**

The Consul Config Watch takes advantage of the ability of consul to [watch a key prefix](https://www.consul.io/docs/agent/watches.html#keyprefix). The Config Watch makes a blocking Consul HTTP API call to determine if any relevant configuration data has changed for the current application. If there is new configuration data a Refresh Event is published. This is equivalent to calling the /refresh actuator endpoint.

To change the frequency of when the Config Watch is called change spring.cloud.consul.config.watch.delay. The default value is 1000, which is in milliseconds. The delay is the amount of time after the end of the previous invocation and the start of the next.

To disable the Config Watch set spring.cloud.consul.config.watch.enabled=false.

The watch uses a Spring TaskScheduler to schedule the call to consul. By default it is a ThreadPoolTaskScheduler with a poolSize of 1. To change the TaskScheduler, create a bean of type TaskScheduler named with the ConsulConfigAutoConfiguration.CONFIG\_WATCH\_TASK\_SCHEDULER\_NAME constant.

**67.4 YAML or Properties with Config**

It may be more convenient to store a blob of properties in YAML or Properties format as opposed to individual key/value pairs. Set the spring.cloud.consul.config.format property to YAML or PROPERTIES. For example to use YAML:

**bootstrap.yml.**

spring:

cloud:

consul:

config:

format: YAML

YAML must be set in the appropriate data key in consul. Using the defaults above the keys would look like:

config/testApp,dev/data

config/testApp/data

config/application,dev/data

config/application/data

You could store a YAML document in any of the keys listed above.

You can change the data key using spring.cloud.consul.config.data-key.

**67.5 git2consul with Config**

git2consul is a Consul community project that loads files from a git repository to individual keys into Consul. By default the names of the keys are names of the files. YAML and Properties files are supported with file extensions of .yml and .properties respectively. Set the spring.cloud.consul.config.format property to FILES. For example:

**bootstrap.yml.**

spring:

cloud:

consul:

config:

format: FILES

Given the following keys in /config, the development profile and an application name of foo:

.gitignore

application.yml

bar.properties

foo-development.properties

foo-production.yml

foo.properties

master.ref

the following property sources would be created:

config/foo-development.properties

config/foo.properties

config/application.yml

The value of each key needs to be a properly formatted YAML or Properties file.

**67.6 Fail Fast**

It may be convenient in certain circumstances (like local development or certain test scenarios) to not fail if consul isn’t available for configuration. Setting spring.cloud.consul.config.failFast=false in bootstrap.yml will cause the configuration module to log a warning rather than throw an exception. This will allow the application to continue startup normally.

**68. Consul Retry**

If you expect that the consul agent may occasionally be unavailable when your app starts, you can ask it to keep trying after a failure. You need to add spring-retryand spring-boot-starter-aop to your classpath. The default behaviour is to retry 6 times with an initial backoff interval of 1000ms and an exponential multiplier of 1.1 for subsequent backoffs. You can configure these properties (and others) using spring.cloud.consul.retry.\* configuration properties. This works with both Spring Cloud Consul Config and Discovery registration.

|  |
| --- |
| [Tip] |
| To take full control of the retry add a @Bean of type RetryOperationsInterceptor with id "consulRetryInterceptor". Spring Retry has a RetryInterceptorBuilder that makes it easy to create one. |

**69. Spring Cloud Bus with Consul**

**69.1 How to activate**

To get started with the Consul Bus use the starter with group org.springframework.cloud and artifact id spring-cloud-starter-consul-bus. See the [Spring Cloud Project page](https://projects.spring.io/spring-cloud/) for details on setting up your build system with the current Spring Cloud Release Train.

See the [Spring Cloud Bus](https://cloud.spring.io/spring-cloud-bus/) documentation for the available actuator endpoints and howto send custom messages.

**70. Circuit Breaker with Hystrix**

Applications can use the Hystrix Circuit Breaker provided by the Spring Cloud Netflix project by including this starter in the projects pom.xml: spring-cloud-starter-hystrix. Hystrix doesn’t depend on the Netflix Discovery Client. The @EnableHystrix annotation should be placed on a configuration class (usually the main class). Then methods can be annotated with @HystrixCommand to be protected by a circuit breaker. See [the documentation](https://projects.spring.io/spring-cloud/spring-cloud.html#_circuit_breaker_hystrix_clients) for more details.

**71. Hystrix metrics aggregation with Turbine and Consul**

Turbine (provided by the Spring Cloud Netflix project), aggregates multiple instances Hystrix metrics streams, so the dashboard can display an aggregate view. Turbine uses the DiscoveryClient interface to lookup relevant instances. To use Turbine with Spring Cloud Consul, configure the Turbine application in a manner similar to the following examples:

**pom.xml.**

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-netflix-turbine</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-consul-discovery</artifactId>

</dependency>

Notice that the Turbine dependency is not a starter. The turbine starter includes support for Netflix Eureka.

**application.yml.**

spring.application.name: turbine

applications: consulhystrixclient

turbine:

aggregator:

clusterConfig: ${applications}

appConfig: ${applications}

The clusterConfig and appConfig sections must match, so it’s useful to put the comma-separated list of service ID’s into a separate configuration property.

**Turbine.java.**

@EnableTurbine

@SpringBootApplication

public class Turbine {

public static void main(String[] args) {

SpringApplication.run(DemoturbinecommonsApplication.class, args);

}

}

**Part X. Spring Cloud Zookeeper**

This project provides Zookeeper integrations for Spring Boot applications through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms. With a few annotations, you can quickly enable and configure the common patterns inside your application and build large distributed systems with Zookeeper based components. The provided patterns include Service Discovery and Configuration. Integration with Spring Cloud Netflix provides Intelligent Routing (Zuul), Client Side Load Balancing (Ribbon), and Circuit Breaker (Hystrix).

**72. Install Zookeeper**

See the [installation documentation](https://zookeeper.apache.org/doc/current/zookeeperStarted.html) for instructions on how to install Zookeeper.

Spring Cloud Zookeeper uses Apache Curator behind the scenes. While Zookeeper 3.5.x is still considered "beta" by the Zookeeper development team, the reality is that it is used in production by many users. However, Zookeeper 3.4.x is also used in production. Prior to Apache Curator 4.0, both versions of Zookeeper were supported via two versions of Apache Curator. Starting with Curator 4.0 both versions of Zookeeper are supported via the same Curator libraries.

In case you are integrating with version 3.4 you need to change the Zookeeper dependency that comes shipped with curator, and thus spring-cloud-zookeeper. To do so simply exclude that dependency and add the 3.4.x version like shown below.

**maven.**

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-zookeeper-all</artifactId>

<exclusions>

<exclusion>

<groupId>org.apache.zookeeper</groupId>

<artifactId>zookeeper</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.apache.zookeeper</groupId>

<artifactId>zookeeper</artifactId>

<version>3.4.12</version>

<exclusions>

<exclusion>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

</exclusion>

</exclusions>

</dependency>

**gradle.**

compile('org.springframework.cloud:spring-cloud-starter-zookeeper-all') {

exclude group: 'org.apache.zookeeper', module: 'zookeeper'

}

compile('org.apache.zookeeper:zookeeper:3.4.12') {

exclude group: 'org.slf4j', module: 'slf4j-log4j12'

}

**73. Service Discovery with Zookeeper**

Service Discovery is one of the key tenets of a microservice based architecture. Trying to hand-configure each client or some form of convention can be difficult to do and can be brittle. [Curator](https://curator.apache.org/)(A Java library for Zookeeper) provides Service Discovery through a [Service Discovery Extension](https://curator.apache.org/curator-x-discovery/). Spring Cloud Zookeeper uses this extension for service registration and discovery.

**73.1 Activating**

Including a dependency on org.springframework.cloud:spring-cloud-starter-zookeeper-discovery enables autoconfiguration that sets up Spring Cloud Zookeeper Discovery.

|  |
| --- |
| [Note] |
| For web functionality, you still need to include org.springframework.boot:spring-boot-starter-web. |
| [Caution] | **Caution** | |
| When working with version 3.4 of Zookeeper you need to change the way you include the dependency as described [here](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-zookeeper-install). | |

**73.2 Registering with Zookeeper**

When a client registers with Zookeeper, it provides metadata (such as host and port, ID, and name) about itself.

The following example shows a Zookeeper client:

*@SpringBootApplication*

*@RestController*

**public** **class** Application {

*@RequestMapping("/")*

**public** String home() {

**return** "Hello world";

}

**public** **static** **void** main(String[] args) {

**new** SpringApplicationBuilder(Application.**class**).web(true).run(args);

}

}

|  |
| --- |
| [Note] |
| The preceding example is a normal Spring Boot application. |

If Zookeeper is located somewhere other than localhost:2181, the configuration must provide the location of the server, as shown in the following example:

**application.yml.**

spring:

cloud:

zookeeper:

connect-string: localhost:2181

|  |  |
| --- | --- |
| [Caution] | **Caution** |
| If you use [Spring Cloud Zookeeper Config](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-zookeeper-config), the values shown in the preceding example need to be in bootstrap.yml instead ofapplication.yml. |

The default service name, instance ID, and port (taken from the Environment) are ${spring.application.name}, the Spring Context ID, and ${server.port}, respectively.

Having spring-cloud-starter-zookeeper-discovery on the classpath makes the app into both a Zookeeper “service” (that is, it registers itself) and a “client” (that is, it can query Zookeeper to locate other services).

If you would like to disable the Zookeeper Discovery Client, you can set spring.cloud.zookeeper.discovery.enabled to false.

**73.3 Using the DiscoveryClient**

Spring Cloud has support for [Feign](https://github.com/spring-cloud/spring-cloud-netflix/blob/master/docs/src/main/asciidoc/spring-cloud-netflix.adoc#spring-cloud-feign) (a REST client builder) and [Spring RestTemplate](https://github.com/spring-cloud/spring-cloud-netflix/blob/master/docs/src/main/asciidoc/spring-cloud-netflix.adoc#spring-cloud-ribbon), using logical service names instead of physical URLs.

You can also use the org.springframework.cloud.client.discovery.DiscoveryClient, which provides a simple API for discovery clients that is not specific to Netflix, as shown in the following example:

*@Autowired*

**private** DiscoveryClient discoveryClient;

**public** String serviceUrl() {

List<ServiceInstance> list = discoveryClient.getInstances("STORES");

**if** (list != null && list.size() > 0 ) {

**return** list.get(0).getUri().toString();

}

**return** null;

}

**74. Using Spring Cloud Zookeeper with Spring Cloud Netflix Components**

Spring Cloud Netflix supplies useful tools that work regardless of which DiscoveryClient implementation you use. Feign, Turbine, Ribbon, and Zuul all work with Spring Cloud Zookeeper.

**74.1 Ribbon with Zookeeper**

Spring Cloud Zookeeper provides an implementation of Ribbon’s ServerList. When you use the spring-cloud-starter-zookeeper-discovery, Ribbon is autoconfigured to use the ZookeeperServerList by default.

**75. Spring Cloud Zookeeper and Service Registry**

Spring Cloud Zookeeper implements the ServiceRegistry interface, letting developers register arbitrary services in a programmatic way.

The ServiceInstanceRegistration class offers a builder() method to create a Registration object that can be used by the ServiceRegistry, as shown in the following example:

*@Autowired*

**private** ZookeeperServiceRegistry serviceRegistry;

**public** **void** registerThings() {

ZookeeperRegistration registration = ServiceInstanceRegistration.builder()

.defaultUriSpec()

.address("anyUrl")

.port(10)

.name("/a/b/c/d/anotherservice")

.build();

**this**.serviceRegistry.register(registration);

}

**75.1 Instance Status**

Netflix Eureka supports having instances that are OUT\_OF\_SERVICE registered with the server. These instances are not returned as active service instances. This is useful for behaviors such as blue/green deployments. (Note that the Curator Service Discovery recipe does not support this behavior.) Taking advantage of the flexible payload has let Spring Cloud Zookeeper implement OUT\_OF\_SERVICE by updating some specific metadata and then filtering on that metadata in the Ribbon ZookeeperServerList. The ZookeeperServerList filters out all non-null instance statuses that do not equal UP. If the instance status field is empty, it is considered to be UP for backwards compatibility. To change the status of an instance, make a POST with OUT\_OF\_SERVICE to the ServiceRegistry instance status actuator endpoint, as shown in the following example:

$ http POST http://localhost:8081/service-registry status=OUT\_OF\_SERVICE

|  |
| --- |
| [Note] |
| The preceding example uses the http command from [https://httpie.org](https://httpie.org/). |

**76. Zookeeper Dependencies**

The following topics cover how to work with Spring Cloud Zookeeper dependencies:

* [Section 76.1, “Using the Zookeeper Dependencies”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-zookeeper-dependencies-using)
* [Section 76.2, “Activating Zookeeper Dependencies”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-zookeeper-dependencies-activating)
* [Section 76.3, “Setting up Zookeeper Dependencies”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-zookeeper-dependencies-setting-up)
* [Section 76.4, “Configuring Spring Cloud Zookeeper Dependencies”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-zookeeper-dependencies-configuring)

**76.1 Using the Zookeeper Dependencies**

Spring Cloud Zookeeper gives you a possibility to provide dependencies of your application as properties. As dependencies, you can understand other applications that are registered in Zookeeper and which you would like to call through [Feign](https://github.com/spring-cloud/spring-cloud-netflix/blob/master/docs/src/main/asciidoc/spring-cloud-netflix.adoc#spring-cloud-feign) (a REST client builder) and [Spring RestTemplate](https://github.com/spring-cloud/spring-cloud-netflix/blob/master/docs/src/main/asciidoc/spring-cloud-netflix.adoc#spring-cloud-ribbon).

You can also use the Zookeeper Dependency Watchers functionality to control and monitor the state of your dependencies.

**76.2 Activating Zookeeper Dependencies**

Including a dependency on org.springframework.cloud:spring-cloud-starter-zookeeper-discovery enables autoconfiguration that sets up Spring Cloud Zookeeper Dependencies. Even if you provide the dependencies in your properties, you can turn off the dependencies. To do so, set thespring.cloud.zookeeper.dependency.enabled property to false (it defaults to true).

**76.3 Setting up Zookeeper Dependencies**

Consider the following example of dependency representation:

**application.yml.**

spring.application.name: yourServiceName

spring.cloud.zookeeper:

dependencies:

newsletter:

path: /path/where/newsletter/has/registered/in/zookeeper

loadBalancerType: ROUND\_ROBIN

contentTypeTemplate: application/vnd.newsletter.$version+json

version: v1

headers:

header1:

- value1

header2:

- value2

required: false

stubs: org.springframework:foo:stubs

mailing:

path: /path/where/mailing/has/registered/in/zookeeper

loadBalancerType: ROUND\_ROBIN

contentTypeTemplate: application/vnd.mailing.$version+json

version: v1

required: true

The next few sections go through each part of the dependency one by one. The root property name is spring.cloud.zookeeper.dependencies.

**76.3.1 Aliases**

Below the root property you have to represent each dependency as an alias. This is due to the constraints of Ribbon, which requires that the application ID be placed in the URL. Consequently, you cannot pass any complex path, suchas /myApp/myRoute/name). The alias is the name you use instead of the serviceId for DiscoveryClient, Feign, or RestTemplate.

In the previous examples, the aliases are newsletter and mailing. The following example shows Feign usage with a newsletter alias:

*@FeignClient("newsletter")*

**public** **interface** NewsletterService {

*@RequestMapping(method = RequestMethod.GET, value = "/newsletter")*

String getNewsletters();

}

**76.3.2 Path**

The path is represented by the path YAML property and is the path under which the dependency is registered under Zookeeper. As described in the [previous section](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-zookeeper-dependencies-setting-up-aliases), Ribbon operates on URLs. As a result, this path is not compliant with its requirement. That is why Spring Cloud Zookeeper maps the alias to the proper path.

**76.3.3 Load Balancer Type**

The load balancer type is represented by loadBalancerType YAML property.

If you know what kind of load-balancing strategy has to be applied when calling this particular dependency, you can provide it in the YAML file, and it is automatically applied. You can choose one of the following load balancing strategies:

* STICKY: Once chosen, the instance is always called.
* RANDOM: Picks an instance randomly.
* ROUND\_ROBIN: Iterates over instances over and over again.

**76.3.4 Content-Type Template and Version**

The Content-Type template and version are represented by the contentTypeTemplate and version YAML properties.

If you version your API in the Content-Type header, you do not want to add this header to each of your requests. Also, if you want to call a new version of the API, you do not want to roam around your code to bump up the API version. That is why you can provide a contentTypeTemplate with a special $version placeholder. That placeholder will be filled by the value of the version YAML property. Consider the following example of a contentTypeTemplate:

application/vnd.newsletter.$version+json

Further consider the following version:

v1

The combination of contentTypeTemplate and version results in the creation of a Content-Type header for each request, as follows:

application/vnd.newsletter.v1+json

**76.3.5 Default Headers**

Default headers are represented by the headers map in YAML.

Sometimes, each call to a dependency requires setting up of some default headers. To not do that in code, you can set them up in the YAML file, as shown in the following example headers section:

headers:

Accept:

- text/html

- application/xhtml+xml

Cache-Control:

- no-cache

That headers section results in adding the Accept and Cache-Control headers with appropriate list of values in your HTTP request.

**76.3.6 Required Dependencies**

Required dependencies are represented by required property in YAML.

If one of your dependencies is required to be up when your application boots, you can set the required: true property in the YAML file.

If your application cannot localize the required dependency during boot time, it throws an exception, and the Spring Context fails to set up. In other words, your application cannot start if the required dependency is not registered in Zookeeper.

You can read more about Spring Cloud Zookeeper Presence Checker [later in this document](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-zookeeper-dependency-watcher-presence-checker).

**76.3.7 Stubs**

You can provide a colon-separated path to the JAR containing stubs of the dependency, as shown in the following example:

stubs: org.springframework:myApp:stubs

where:

* org.springframework is the groupId.
* myApp is the artifactId.
* stubs is the classifier. (Note that stubs is the default value.)

Because stubs is the default classifier, the preceding example is equal to the following example:

stubs: org.springframework:myApp

**76.4 Configuring Spring Cloud Zookeeper Dependencies**

You can set the following properties to enable or disable parts of Zookeeper Dependencies functionalities:

* spring.cloud.zookeeper.dependencies: If you do not set this property, you cannot use Zookeeper Dependencies.
* spring.cloud.zookeeper.dependency.ribbon.enabled (enabled by default): Ribbon requires either explicit global configuration or a particular one for a dependency. By turning on this property, runtime load balancing strategy resolution is possible, and you can use the loadBalancerType section of the Zookeeper Dependencies. The configuration that needs this property has an implementation of LoadBalancerClient that delegates to the ILoadBalancer presented in the next bullet.
* spring.cloud.zookeeper.dependency.ribbon.loadbalancer (enabled by default): Thanks to this property, the custom ILoadBalancer knows that the part of the URI passed to Ribbon might actually be the alias that has to be resolved to a proper path in Zookeeper. Without this property, you cannot register applications under nested paths.
* spring.cloud.zookeeper.dependency.headers.enabled (enabled by default): This property registers a RibbonClient that automatically appends appropriate headers and content types with their versions, as presented in the Dependency configuration. Without this setting, those two parameters do not work.
* spring.cloud.zookeeper.dependency.resttemplate.enabled (enabled by default): When enabled, this property modifies the request headers of a @LoadBalanced-annotated RestTemplate such that it passes headers and content type with the version set in dependency configuration. Without this setting, those two parameters do not work.

**77. Spring Cloud Zookeeper Dependency Watcher**

The Dependency Watcher mechanism lets you register listeners to your dependencies. The functionality is, in fact, an implementation of the Observator pattern. When a dependency changes, its state (to either UP or DOWN), some custom logic can be applied.

**77.1 Activating**

Spring Cloud Zookeeper Dependencies functionality needs to be enabled for you to use the Dependency Watcher mechanism.

**77.2 Registering a Listener**

To register a listener, you must implement an interface called org.springframework.cloud.zookeeper.discovery.watcher.DependencyWatcherListener and register it as a bean. The interface gives you one method:

**void** stateChanged(String dependencyName, DependencyState newState);

If you want to register a listener for a particular dependency, the dependencyName would be the discriminator for your concrete implementation. newState provides you with information about whether your dependency has changed to CONNECTED or DISCONNECTED.

**77.3 Using the Presence Checker**

Bound with the Dependency Watcher is the functionality called Presence Checker. It lets you provide custom behavior when your application boots, to react according to the state of your dependencies.

The default implementation of the abstract org.springframework.cloud.zookeeper.discovery.watcher.presence.DependencyPresenceOnStartupVerifierclass is the org.springframework.cloud.zookeeper.discovery.watcher.presence.DefaultDependencyPresenceOnStartupVerifier, which works in the following way.

1. If the dependency is marked us required and is not in Zookeeper, when your application boots, it throws an exception and shuts down.
2. If the dependency is not required, the org.springframework.cloud.zookeeper.discovery.watcher.presence.LogMissingDependencyChecker logs that the dependency is missing at the WARN level.

Because the DefaultDependencyPresenceOnStartupVerifier is registered only when there is no bean of type DependencyPresenceOnStartupVerifier, this functionality can be overridden.

**78. Distributed Configuration with Zookeeper**

Zookeeper provides a [hierarchical namespace](https://zookeeper.apache.org/doc/current/zookeeperOver.html#sc_dataModelNameSpace) that lets clients store arbitrary data, such as configuration data. Spring Cloud Zookeeper Config is an alternative to the[Config Server and Client](https://github.com/spring-cloud/spring-cloud-config). Configuration is loaded into the Spring Environment during the special “bootstrap” phase. Configuration is stored in the /config namespace by default. Multiple PropertySource instances are created, based on the application’s name and the active profiles, to mimic the Spring Cloud Config order of resolving properties. For example, an application with a name of testApp and with the dev profile has the following property sources created for it:

* config/testApp,dev
* config/testApp
* config/application,dev
* config/application

The most specific property source is at the top, with the least specific at the bottom. Properties in the config/application namespace apply to all applications that use zookeeper for configuration. Properties in the config/testApp namespace are available only to the instances of the service named testApp.

Configuration is currently read on startup of the application. Sending a HTTP POST request to /refresh causes the configuration to be reloaded. Watching the configuration namespace (which Zookeeper supports) is not currently implemented.

**78.1 Activating**

Including a dependency on org.springframework.cloud:spring-cloud-starter-zookeeper-config enables autoconfiguration that sets up Spring Cloud Zookeeper Config.

|  |  |
| --- | --- |
| [Caution] | **Caution** |
| When working with version 3.4 of Zookeeper you need to change the way you include the dependency as described [here](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-zookeeper-install). |

**78.2 Customizing**

Zookeeper Config may be customized by setting the following properties:

**bootstrap.yml.**

spring:

cloud:

zookeeper:

config:

enabled: true

root: configuration

defaultContext: apps

profileSeparator: '::'

* enabled: Setting this value to false disables Zookeeper Config.
* root: Sets the base namespace for configuration values.
* defaultContext: Sets the name used by all applications.
* profileSeparator: Sets the value of the separator used to separate the profile name in property sources with profiles.

**78.3 Access Control Lists (ACLs)**

You can add authentication information for Zookeeper ACLs by calling the addAuthInfo method of a CuratorFramework bean. One way to accomplish this is to provide your own CuratorFramework bean, as shown in the following example:

*@BoostrapConfiguration*

**public** **class** CustomCuratorFrameworkConfig {

*@Bean*

**public** CuratorFramework curatorFramework() {

CuratorFramework curator = **new** CuratorFramework();

curator.addAuthInfo("digest", "user:password".getBytes());

**return** curator;

}

}

Consult [the ZookeeperAutoConfiguration class](https://github.com/spring-cloud/spring-cloud-zookeeper/blob/master/spring-cloud-zookeeper-core/src/main/java/org/springframework/cloud/zookeeper/ZookeeperAutoConfiguration.java) to see how the CuratorFramework bean’s default configuration.

Alternatively, you can add your credentials from a class that depends on the existing CuratorFramework bean, as shown in the following example:

*@BoostrapConfiguration*

**public** **class** DefaultCuratorFrameworkConfig {

**public** ZookeeperConfig(CuratorFramework curator) {

curator.addAuthInfo("digest", "user:password".getBytes());

}

}

The creation of this bean must occur during the boostrapping phase. You can register configuration classes to run during this phase by annotating them with@BootstrapConfiguration and including them in a comma-separated list that you set as the value of the org.springframework.cloud.bootstrap.BootstrapConfiguration property in the resources/META-INF/spring.factories file, as shown in the following example:

**resources/META-INF/spring.factories.**

org.springframework.cloud.bootstrap.BootstrapConfiguration=\

my.project.CustomCuratorFrameworkConfig,\

my.project.DefaultCuratorFrameworkConfig

**Part XI. Spring Boot Cloud CLI**

Spring Boot CLI provides [Spring Boot](https://projects.spring.io/spring-boot) command line features for [Spring Cloud](https://github.com/spring-cloud). You can write Groovy scripts to run Spring Cloud component applications (e.g. @EnableEurekaServer). You can also easily do things like encryption and decryption to support Spring Cloud Config clients with secret configuration values. With the Launcher CLI you can launch services like Eureka, Zipkin, Config Server conveniently all at once from the command line (very useful at development time).

|  |
| --- |
| [Note] |
| Spring Cloud is released under the non-restrictive Apache 2.0 license. If you would like to contribute to this section of the documentation or if you find an error, please find the source code and issue trackers in the project at [github](https://github.com/spring-cloud/spring-cloud-cli/tree/master/docs/src/main/asciidoc). |

**79. Installation**

To install, make sure you have [Spring Boot CLI](https://github.com/spring-projects/spring-boot) (1.5.2 or better):

$ spring version

Spring CLI v1.5.4.RELEASE

E.g. for SDKMan users

$ sdk install springboot 1.5.4.RELEASE

$ sdk use springboot 1.5.4.RELEASE

and install the Spring Cloud plugin

$ mvn install

$ spring install org.springframework.cloud:spring-cloud-cli:1.4.0.BUILD-SNAPSHOT

|  |  |
| --- | --- |
| [Important] | **Important** |
| **Prerequisites:** to use the encryption and decryption features you need the full-strength JCE installed in your JVM (it’s not there by default). You can download the "Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files" from Oracle, and follow instructions for installation (essentially replace the 2 policy files in the JRE lib/security directory with the ones that you downloaded). |

**80. Running Spring Cloud Services in Development**

The Launcher CLI can be used to run common services like Eureka, Config Server etc. from the command line. To list the available services you can do spring cloud --list, and to launch a default set of services just spring cloud. To choose the services to deploy, just list them on the command line, e.g.

$ spring cloud eureka configserver h2 kafka stubrunner zipkin

Summary of supported deployables:

| **Service** | **Name** | **Address** | **Description** |
| --- | --- | --- | --- |
| eureka | Eureka Server | [http://localhost:8761](http://localhost:8761/) | Eureka server for service registration and discovery. All the other services show up in its catalog by default. |
| configserver | Config Server | [http://localhost:8888](http://localhost:8888/) | Spring Cloud Config Server running in the "native" profile and serving configuration from the local directory ./launcher |
| h2 | H2 Database | [http://localhost:9095](http://localhost:9095/) (console), jdbc:h2:tcp://localhost:9096/{data} | Relation database service. Use a file path for {data} (e.g. ./target/test) when you connect. Remember that you can add ;MODE=MYSQL or ;MODE=POSTGRESQL to connect with compatibility to other server types. |
| kafka | Kafka Broker | [http://localhost:9091](http://localhost:9091/) (actuator endpoints), localhost:9092 |  |
| hystrixdashboard | Hystrix Dashboard | [http://localhost:7979](http://localhost:7979/) | Any Spring Cloud app that declares Hystrix circuit breakers publishes metrics on /hystrix.stream. Type that address into the dashboard to visualize all the metrics, |
| dataflow | Dataflow Server | [http://localhost:9393](http://localhost:9393/) | Spring Cloud Dataflow server with UI at /admin-ui. Connect the Dataflow shell to target at root path. |
| zipkin | Zipkin Server | [http://localhost:9411](http://localhost:9411/) | Zipkin Server with UI for visualizing traces. Stores span data in memory and accepts them via HTTP POST of JSON data. |
| stubrunner | Stub Runner Boot | [http://localhost:8750](http://localhost:8750/) | Downloads WireMock stubs, starts WireMock and feeds the started servers with stored stubs. Pass stubrunner.ids to pass stub coordinates and then go to <http://localhost:8750/stubs>. |

Each of these apps can be configured using a local YAML file with the same name (in the current working directory or a subdirectory called "config" or in ~/.spring-cloud). E.g. in configserver.yml you might want to do something like this to locate a local git repository for the backend:

**configserver.yml.**

spring:

profiles:

active: git

cloud:

config:

server:

git:

uri: file://${user.home}/dev/demo/config-repo

E.g. in Stub Runner app you could fetch stubs from your local .m2 in the following way.

**stubrunner.yml.**

stubrunner:

workOffline: **true**

ids:

- com.example:beer-api-producer:+:9876

**80.1 Adding Additional Applications**

Additional applications can be added to ./config/cloud.yml (not ./config.yml because that would replace the defaults), e.g. with

**config/cloud.yml.**

spring:

cloud:

launcher:

deployables:

source:

coordinates: maven://com.example:source:0.0.1-SNAPSHOT

port: 7000

sink:

coordinates: maven://com.example:sink:0.0.1-SNAPSHOT

port: 7001

when you list the apps:

$ spring cloud --list

source sink configserver dataflow eureka h2 hystrixdashboard kafka stubrunner zipkin

(notice the additional apps at the start of the list).

**81. Writing Groovy Scripts and Running Applications**

Spring Cloud CLI has support for most of the Spring Cloud declarative features, such as the @Enable\* class of annotations. For example, here is a fully functional Eureka server

**app.groovy.**

*@EnableEurekaServer*

**class** Eureka {}

which you can run from the command line like this

$ spring run app.groovy

To include additional dependencies, often it suffices just to add the appropriate feature-enabling annotation, e.g. @EnableConfigServer, @EnableOAuth2Sso or @EnableEurekaClient. To manually include a dependency you can use a @Grab with the special "Spring Boot" short style artifact co-ordinates, i.e. with just the artifact ID (no need for group or version information), e.g. to set up a client app to listen on AMQP for management events from the Spring CLoud Bus:

**app.groovy.**

*@Grab('spring-cloud-starter-bus-amqp')*

*@RestController*

**class** Service {

*@RequestMapping('/')*

def home() { [message: 'Hello'] }

}

**82. Encryption and Decryption**

The Spring Cloud CLI comes with an "encrypt" and a "decrypt" command. Both accept arguments in the same form with a key specified as a mandatory "--key", e.g.

$ spring encrypt mysecret --key foo

682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda

$ spring decrypt --key foo 682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda

mysecret

To use a key in a file (e.g. an RSA public key for encyption) prepend the key value with "@" and provide the file path, e.g.

$ spring encrypt mysecret --key @${HOME}/.ssh/id\_rsa.pub

AQAjPgt3eFZQXwt8tsHAVv/QHiY5sI2dRcR+...

**Part XII. Spring Cloud Security**

Spring Cloud Security offers a set of primitives for building secure applications and services with minimum fuss. A declarative model which can be heavily configured externally (or centrally) lends itself to the implementation of large systems of co-operating, remote components, usually with a central indentity management service. It is also extremely easy to use in a service platform like Cloud Foundry. Building on Spring Boot and Spring Security OAuth2 we can quickly create systems that implement common patterns like single sign on, token relay and token exchange.

|  |
| --- |
| [Note] |
| Spring Cloud is released under the non-restrictive Apache 2.0 license. If you would like to contribute to this section of the documentation or if you find an error, please find the source code and issue trackers in the project at [github](https://github.com/spring-cloud/spring-cloud-security/tree/master/src/main/asciidoc). |

**83. Quickstart**

**83.1 OAuth2 Single Sign On**

Here’s a Spring Cloud "Hello World" app with HTTP Basic authentication and a single user account:

**app.groovy.**

*@Grab('spring-boot-starter-security')*

*@Controller*

**class** Application {

*@RequestMapping('/')*

String home() {

'Hello World'

}

}

You can run it with spring run app.groovy and watch the logs for the password (username is "user"). So far this is just the default for a Spring Boot app.

Here’s a Spring Cloud app with OAuth2 SSO:

**app.groovy.**

*@Controller*

*@EnableOAuth2Sso*

**class** Application {

*@RequestMapping('/')*

String home() {

'Hello World'

}

}

Spot the difference? This app will actually behave exactly the same as the previous one, because it doesn’t know it’s OAuth2 credentals yet.

You can register an app in github quite easily, so try that if you want a production app on your own domain. If you are happy to test on localhost:8080, then set up these properties in your application configuration:

**application.yml.**

security:

oauth2:

client:

clientId: bd1c0a783ccdd1c9b9e4

clientSecret: 1a9030fbca47a5b2c28e92f19050bb77824b5ad1

accessTokenUri: https://github.com/login/oauth/access\_token

userAuthorizationUri: https://github.com/login/oauth/authorize

clientAuthenticationScheme: form

resource:

userInfoUri: https://api.github.com/user

preferTokenInfo: **false**

run the app above and it will redirect to github for authorization. If you are already signed into github you won’t even notice that it has authenticated. These credentials will only work if your app is running on port 8080.

To limit the scope that the client asks for when it obtains an access token you can set security.oauth2.client.scope (comma separated or an array in YAML). By default the scope is empty and it is up to to Authorization Server to decide what the defaults should be, usually depending on the settings in the client registration that it holds.

|  |
| --- |
| [Note] |
| The examples above are all Groovy scripts. If you want to write the same code in Java (or Groovy) you need to add Spring Security OAuth2 to the classpath (e.g. see the [sample here](https://github.com/spring-cloud-samples/sso)). |

**83.2 OAuth2 Protected Resource**

You want to protect an API resource with an OAuth2 token? Here’s a simple example (paired with the client above):

**app.groovy.**

*@Grab('spring-cloud-starter-security')*

*@RestController*

*@EnableResourceServer*

**class** Application {

*@RequestMapping('/')*

def home() {

[message: 'Hello World']

}

}

and

**application.yml.**

security:

oauth2:

resource:

userInfoUri: https://api.github.com/user

preferTokenInfo: **false**

**84. More Detail**

**84.1 Single Sign On**

|  |
| --- |
| [Note] |
| All of the OAuth2 SSO and resource server features moved to Spring Boot in version 1.3. You can find documentation in the [Spring Boot user guide](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/). |

**84.2 Token Relay**

A Token Relay is where an OAuth2 consumer acts as a Client and forwards the incoming token to outgoing resource requests. The consumer can be a pure Client (like an SSO application) or a Resource Server.

**84.2.1 Client Token Relay**

If your app is a user facing OAuth2 client (i.e. has declared @EnableOAuth2Sso or @EnableOAuth2Client) then it has an OAuth2ClientContext in request scope from Spring Boot. You can create your own OAuth2RestTemplate from this context and an autowired OAuth2ProtectedResourceDetails, and then the context will always forward the access token downstream, also refreshing the access token automatically if it expires. (These are features of Spring Security and Spring Boot.)

|  |
| --- |
| [Note] |
| Spring Boot (1.4.1) does not create an OAuth2ProtectedResourceDetails automatically if you are using client\_credentials tokens. In that case you need to create your own ClientCredentialsResourceDetails and configure it with @ConfigurationProperties("security.oauth2.client"). |

**84.2.2 Client Token Relay in Zuul Proxy**

If your app also has a [Spring Cloud Zuul](https://cloud.spring.io/spring-cloud.html#netflix-zuul-reverse-proxy) embedded reverse proxy (using @EnableZuulProxy) then you can ask it to forward OAuth2 access tokens downstream to the services it is proxying. Thus the SSO app above can be enhanced simply like this:

**app.groovy.**

*@Controller*

*@EnableOAuth2Sso*

*@EnableZuulProxy*

**class** Application {

}

and it will (in addition to logging the user in and grabbing a token) pass the authentication token downstream to the /proxy/\* services. If those services are implemented with @EnableResourceServer then they will get a valid token in the correct header.

How does it work? The @EnableOAuth2Sso annotation pulls in spring-cloud-starter-security (which you could do manually in a traditional app), and that in turn triggers some autoconfiguration for a ZuulFilter, which itself is activated because Zuul is on the classpath (via @EnableZuulProxy). The [filter](https://github.com/spring-cloud/spring-cloud-security/tree/master/src/main/java/org/springframework/cloud/security/oauth2/proxy/OAuth2TokenRelayFilter.java) just extracts an access token from the currently authenticated user, and puts it in a request header for the downstream requests.

**84.2.3 Resource Server Token Relay**

If your app has @EnableResourceServer you might want to relay the incoming token downstream to other services. If you use a RestTemplate to contact the downstream services then this is just a matter of how to create the template with the right context.

If your service uses UserInfoTokenServices to authenticate incoming tokens (i.e. it is using the security.oauth2.user-info-uri configuration), then you can simply create an OAuth2RestTemplate using an autowired OAuth2ClientContext (it will be populated by the authentication process before it hits the backend code). Equivalently (with Spring Boot 1.4), you could inject a UserInfoRestTemplateFactory and grab its OAuth2RestTemplate in your configuration. For example:

**MyConfiguration.java.**

*@Bean*

**public** OAuth2RestTemplate restTemplate(UserInfoRestTemplateFactory factory) {

**return** factory.getUserInfoRestTemplate();

}

This rest template will then have the same OAuth2ClientContext (request-scoped) that is used by the authentication filter, so you can use it to send requests with the same access token.

If your app is not using UserInfoTokenServices but is still a client (i.e. it declares @EnableOAuth2Client or @EnableOAuth2Sso), then with Spring Security Cloud any OAuth2RestOperations that the user creates from an @Autowired @OAuth2Context will also forward tokens. This feature is implemented by default as an MVC handler interceptor, so it only works in Spring MVC. If you are not using MVC you could use a custom filter or AOP interceptor wrapping an AccessTokenContextRelayto provide the same feature.

Here’s a basic example showing the use of an autowired rest template created elsewhere ("foo.com" is a Resource Server accepting the same tokens as the surrounding app):

**MyController.java.**

*@Autowired*

**private** OAuth2RestOperations restTemplate;

*@RequestMapping("/relay")*

**public** String relay() {

ResponseEntity<String> response =

restTemplate.getForEntity("https://foo.com/bar", String.**class**);

**return** "Success! (" + response.getBody() + ")";

}

If you don’t want to forward tokens (and that is a valid choice, since you might want to act as yourself, rather than the client that sent you the token), then you only need to create your own OAuth2Context instead of autowiring the default one.

Feign clients will also pick up an interceptor that uses the OAuth2ClientContext if it is available, so they should also do a token relay anywhere where a RestTemplate would.

**85. Configuring Authentication Downstream of a Zuul Proxy**

You can control the authorization behaviour downstream of an @EnableZuulProxy through the proxy.auth.\* settings. Example:

**application.yml.**

proxy:

auth:

routes:

customers: oauth2

stores: passthru

recommendations: none

In this example the "customers" service gets an OAuth2 token relay, the "stores" service gets a passthrough (the authorization header is just passed downstream), and the "recommendations" service has its authorization header removed. The default behaviour is to do a token relay if there is a token available, and passthru otherwise.

See [ProxyAuthenticationProperties](https://github.com/spring-cloud/spring-cloud-security/tree/master/src/main/java/org/springframework/cloud/security/oauth2/proxy/ProxyAuthenticationProperties) for full details.

**Part XIII. Spring Cloud for Cloud Foundry**

Spring Cloud for Cloudfoundry makes it easy to run [Spring Cloud](https://github.com/spring-cloud) apps in [Cloud Foundry](https://github.com/cloudfoundry) (the Platform as a Service). Cloud Foundry has the notion of a "service", which is middlware that you "bind" to an app, essentially providing it with an environment variable containing credentials (e.g. the location and username to use for the service).

The spring-cloud-cloudfoundry-commons module configures the Reactor-based Cloud Foundry Java client, v 3.0, and can be used standalone.

The spring-cloud-cloudfoundry-web project provides basic support for some enhanced features of webapps in Cloud Foundry: binding automatically to single-sign-on services and optionally enabling sticky routing for discovery.

The spring-cloud-cloudfoundry-discovery project provides an implementation of Spring Cloud Commons DiscoveryClient so you can@EnableDiscoveryClient and provide your credentials as spring.cloud.cloudfoundry.discovery.[username,password] (also \*.url if you are not connecting to [Pivotal Web Services](https://run.pivotal.io/)) and then you can use the DiscoveryClient directly or via a LoadBalancerClient.

The first time you use it the discovery client might be slow owing to the fact that it has to get an access token from Cloud Foundry.

**86. Discovery**

Here’s a Spring Cloud app with Cloud Foundry discovery:

**app.groovy.**

*@Grab('org.springframework.cloud:spring-cloud-cloudfoundry')*

*@RestController*

*@EnableDiscoveryClient*

**class** Application {

*@Autowired*

DiscoveryClient client

*@RequestMapping('/')*

String home() {

'Hello from ' + client.getLocalServiceInstance()

}

}

If you run it without any service bindings:

$ spring jar app.jar app.groovy

$ cf push -p app.jar

It will show its app name in the home page.

The DiscoveryClient can lists all the apps in a space, according to the credentials it is authenticated with, where the space defaults to the one the client is running in (if any). If neither org nor space are configured, they default per the user’s profile in Cloud Foundry.

**87. Single Sign On**

|  |
| --- |
| [Note] |
| All of the OAuth2 SSO and resource server features moved to Spring Boot in version 1.3. You can find documentation in the [Spring Boot user guide](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/). |

This project provides automatic binding from CloudFoundry service credentials to the Spring Boot features. If you have a CloudFoundry service called "sso", for instance, with credentials containing "client\_id", "client\_secret" and "auth\_domain", it will bind automatically to the Spring OAuth2 client that you enable with @EnableOAuth2Sso(from Spring Boot). The name of the service can be parameterized using spring.oauth2.sso.serviceId.

**Part XIV. Spring Cloud Contract**

*Documentation Authors: Adam Dudczak, Mathias Düsterhöft, Marcin Grzejszczak, Dennis Kieselhorst, Jakub Kubryński, Karol Lassak, Olga Maciaszek-Sharma, Mariusz Smykuła, Dave Syer, Jay Bryant*

Finchley.SR1

**88. Spring Cloud Contract**

You need confidence when pushing new features to a new application or service in a distributed system. This project provides support for Consumer Driven Contracts and service schemas in Spring applications (for both HTTP and message-based interactions), covering a range of options for writing tests, publishing them as assets, and asserting that a contract is kept by producers and consumers.

**89. Spring Cloud Contract Verifier Introduction**

|  |
| --- |
| [Tip] |
| The Accurest project was initially started by Marcin Grzejszczak and Jakub Kubrynski ([codearte.io](http://codearte.io/)) |

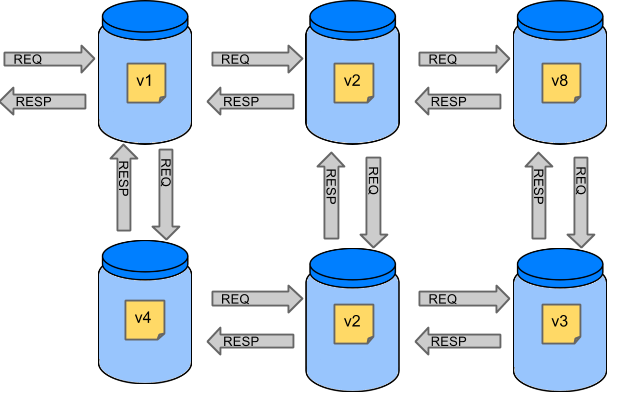
Spring Cloud Contract Verifier enables Consumer Driven Contract (CDC) development of JVM-based applications. It moves TDD to the level of software architecture.

Spring Cloud Contract Verifier ships with *Contract Definition Language* (CDL). Contract definitions are used to produce the following resources:

* JSON stub definitions to be used by WireMock when doing integration testing on the client code (*client tests*). Test code must still be written by hand, and test data is produced by Spring Cloud Contract Verifier.
* Messaging routes, if you’re using a messaging service. We integrate with Spring Integration, Spring Cloud Stream, Spring AMQP, and Apache Camel. You can also set your own integrations.
* Acceptance tests (in JUnit or Spock) are used to verify if server-side implementation of the API is compliant with the contract (*server tests*). A full test is generated by Spring Cloud Contract Verifier.

**89.1 Why a Contract Verifier?**

Assume that we have a system consisting of multiple microservices:



**89.1.1 Testing issues**

If we wanted to test the application in top left corner to determine whether it can communicate with other services, we could do one of two things:

* Deploy all microservices and perform end-to-end tests.
* Mock other microservices in unit/integration tests.

Both have their advantages but also a lot of disadvantages.

**Deploy all microservices and perform end to end tests**

Advantages:

* Simulates production.
* Tests real communication between services.

Disadvantages:

* To test one microservice, we have to deploy 6 microservices, a couple of databases, etc.
* The environment where the tests run is locked for a single suite of tests (nobody else would be able to run the tests in the meantime).
* They take a long time to run.
* The feedback comes very late in the process.
* They are extremely hard to debug.

**Mock other microservices in unit/integration tests**

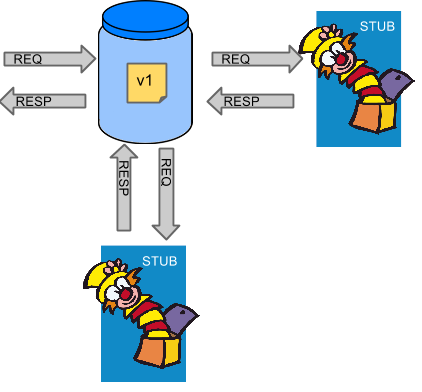
Advantages:

* They provide very fast feedback.
* They have no infrastructure requirements.

Disadvantages:

* The implementor of the service creates stubs that might have nothing to do with reality.
* You can go to production with passing tests and failing production.

To solve the aforementioned issues, Spring Cloud Contract Verifier with Stub Runner was created. The main idea is to give you very fast feedback, without the need to set up the whole world of microservices. If you work on stubs, then the only applications you need are those that your application directly uses.



Spring Cloud Contract Verifier gives you the certainty that the stubs that you use were created by the service that you’re calling. Also, if you can use them, it means that they were tested against the producer’s side. In short, you can trust those stubs.

**89.2 Purposes**

The main purposes of Spring Cloud Contract Verifier with Stub Runner are:

* To ensure that WireMock/Messaging stubs (used when developing the client) do exactly what the actual server-side implementation does.
* To promote ATDD method and Microservices architectural style.
* To provide a way to publish changes in contracts that are immediately visible on both sides.
* To generate boilerplate test code to be used on the server side.

|  |  |
| --- | --- |
| [Important] | **Important** |
| Spring Cloud Contract Verifier’s purpose is NOT to start writing business features in the contracts. Assume that we have a business use case of fraud check. If a user can be a fraud for 100 different reasons, we would assume that you would create 2 contracts, one for the positive case and one for the negative case. Contract tests are used to test contracts between applications and not to simulate full behavior. |

**89.3 How It Works**

This section explores how Spring Cloud Contract Verifier with Stub Runner works.

**89.3.1 A Three-second Tour**

This very brief tour walks through using Spring Cloud Contract:

* [the section called “On the Producer Side”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-contract-verifier-intro-three-second-tour-producer)
* [the section called “On the Consumer Side”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-contract-verifier-intro-three-second-tour-consumer)

You can find a somewhat longer tour [here](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-contract-verifier-intro-three-minute-tour).

**On the Producer Side**

To start working with Spring Cloud Contract, add files with REST/ messaging contracts expressed in either Groovy DSL or YAML to the contracts directory, which is set by the contractsDslDir property. By default, it is $rootDir/src/test/resources/contracts.

Then add the Spring Cloud Contract Verifier dependency and plugin to your build file, as shown in the following example:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-contract-verifier</artifactId>

<scope>test</scope>

</dependency>

The following listing shows how to add the plugin, which should go in the build/plugins portion of the file:

<plugin>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions>

</plugin>

Running ./mvnw clean install automatically generates tests that verify the application compliance with the added contracts. By default, the tests get generated under org.springframework.cloud.contract.verifier.tests..

As the implementation of the functionalities described by the contracts is not yet present, the tests fail.

To make them pass, you must add the correct implementation of either handling HTTP requests or messages. Also, you must add a correct base test class for auto-generated tests to the project. This class is extended by all the auto-generated tests, and it should contain all the setup necessary to run them (for example RestAssuredMockMvc controller setup or messaging test setup).

Once the implementation and the test base class are in place, the tests pass, and both the application and the stub artifacts are built and installed in the local Maven repository. The changes can now be merged, and both the application and the stub artifacts may be published in an online repository.

**On the Consumer Side**

Spring Cloud Contract Stub Runner can be used in the integration tests to get a running WireMock instance or messaging route that simulates the actual service.

To do so, add the dependency to Spring Cloud Contract Stub Runner, as shown in the following example:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-contract-stub-runner</artifactId>

<scope>test</scope>

</dependency>

You can get the Producer-side stubs installed in your Maven repository in either of two ways:

* By checking out the Producer side repository and adding contracts and generating the stubs by running the following commands:
* $ **cd** local-http-server-repo

$ ./mvnw clean install -DskipTests

|  |
| --- |
| [Tip] |
| The tests are being skipped because the Producer-side contract implementation is not in place yet, so the automatically-generated contract tests fail. |

* By getting already-existing producer service stubs from a remote repository. To do so, pass the stub artifact IDs and artifact repository URL as Spring Cloud Contract Stub Runner properties, as shown in the following example:
* stubrunner:
* ids: 'com.example:http-server-dsl:+:stubs:8080'

repositoryRoot: http://repo.spring.io/libs-snapshot

Now you can annotate your test class with @AutoConfigureStubRunner. In the annotation, provide the group-id and artifact-id values for Spring Cloud Contract Stub Runner to run the collaborators' stubs for you, as shown in the following example:

*@RunWith(SpringRunner.class)*

*@SpringBootTest(webEnvironment=WebEnvironment.NONE)*

*@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:6565"},*

*stubsMode = StubRunnerProperties.StubsMode.LOCAL)*

*@DirtiesContext*

**public** **class** LoanApplicationServiceTests {

|  |
| --- |
| [Tip] |
| Use the REMOTE stubsMode when downloading stubs from an online repository and LOCAL for offline work. |

Now, in your integration test, you can receive stubbed versions of HTTP responses or messages that are expected to be emitted by the collaborator service.

**89.3.2 A Three-minute Tour**

This brief tour walks through using Spring Cloud Contract:

* [the section called “On the Producer Side”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-contract-verifier-intro-three-minute-tour-producer)
* [the section called “On the Consumer Side”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-contract-verifier-intro-three-minute-tour-consumer)

You can find an even more brief tour [here](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#spring-cloud-contract-verifier-intro-three-second-tour).

**On the Producer Side**

To start working with Spring Cloud Contract, add files with REST/ messaging contracts expressed in either Groovy DSL or YAML to the contracts directory, which is set by the contractsDslDir property. By default, it is $rootDir/src/test/resources/contracts.

For the HTTP stubs, a contract defines what kind of response should be returned for a given request (taking into account the HTTP methods, URLs, headers, status codes, and so on). The following example shows how an HTTP stub contract in Groovy DSL:

**package** contracts

org.springframework.cloud.contract.spec.Contract.make {

request {

method 'PUT'

url '/fraudcheck'

body([

"client.id": $(regex('[0-9]{10}')),

loanAmount: 99999

])

headers {

contentType('application/json')

}

}

response {

status OK()

body([

fraudCheckStatus: "FRAUD",

"rejection.reason": "Amount too high"

])

headers {

contentType('application/json')

}

}

}

The same contract expressed in YAML would look like the following example:

request:

method: PUT

url: /fraudcheck

body:

"client.id": 1234567890

loanAmount: 99999

headers:

Content-Type: application/json

matchers:

body:

- path: $.['client.id'**]**

type: by\_regex

value: "[0-9]{10}"

response:

status: 200

body:

fraudCheckStatus: "FRAUD"

"rejection.reason": "Amount too high"

headers:

Content-Type: application/json;charset=UTF-8

In the case of messaging, you can define:

* The input and the output messages can be defined (taking into account from and where it was sent, the message body, and the header).
* The methods that should be called after the message is received.
* The methods that, when called, should trigger a message.

The following example shows a Camel messaging contract expressed in Groovy DSL:

def contractDsl = Contract.make {

label 'some\_label'

input {

messageFrom('jms:delete')

messageBody([

bookName: 'foo'

])

messageHeaders {

header('sample', 'header')

}

assertThat('bookWasDeleted()')

}

}

The following example shows the same contract expressed in YAML:

label: some\_label

input:

messageFrom: jms:delete

messageBody:

bookName: 'foo'

messageHeaders:

sample: header

assertThat: bookWasDeleted()

Then you can add Spring Cloud Contract Verifier dependency and plugin to your build file, as shown in the following example:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-contract-verifier</artifactId>

<scope>test</scope>

</dependency>

The following listing shows how to add the plugin, which should go in the build/plugins portion of the file:

<plugin>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions>

</plugin>

Running ./mvnw clean install automatically generates tests that verify the application compliance with the added contracts. By default, the generated tests are under org.springframework.cloud.contract.verifier.tests..

The following example shows a sample auto-generated test for an HTTP contract:

*@Test*

**public** **void** validate\_shouldMarkClientAsFraud() **throws** Exception {

*// given:*

MockMvcRequestSpecification request = given()

.header("Content-Type", "application/vnd.fraud.v1+json")

.body("{\"client.id\":\"1234567890\",\"loanAmount\":99999}");

*// when:*

ResponseOptions response = given().spec(request)

.put("/fraudcheck");

*// then:*

assertThat(response.statusCode()).isEqualTo(200);

assertThat(response.header("Content-Type")).matches("application/vnd.fraud.v1.json.\*");

*// and:*

DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());

assertThatJson(parsedJson).field("['fraudCheckStatus']").matches("[A-Z]{5}");

assertThatJson(parsedJson).field("['rejection.reason']").isEqualTo("Amount too high");

}

The preceding example uses Spring’s MockMvc to run the tests. This is the default test mode for HTTP contracts. However, JAX-RX client and explicit HTTP invocations can also be used. (To do so, change the testMode property of the plugin to JAX-RS or EXPLICIT, respectively.)

Apart from the default JUnit, you can instead use Spock tests, by setting the plugin testFramework property to Spock.

|  |
| --- |
| [Tip] |
| You can now also generate WireMock scenarios based on the contracts, by including an order number followed by an underscore at the beginning of the contract file names. |

The following example shows an auto-generated test in Spock for a messaging stub contract:

[source,groovy,indent=0]

given:

ContractVerifierMessage inputMessage = contractVerifierMessaging.create(

\'\'\'{"bookName":"foo"}\'\'\',

['sample': 'header']

)

when:

contractVerifierMessaging.send(inputMessage, 'jms:delete')

then:

noExceptionThrown()

bookWasDeleted()

As the implementation of the functionalities described by the contracts is not yet present, the tests fail.

To make them pass, you must add the correct implementation of handling either HTTP requests or messages. Also, you must add a correct base test class for auto-generated tests to the project. This class is extended by all the auto-generated tests and should contain all the setup necessary to run them (for example, RestAssuredMockMvc controller setup or messaging test setup).

Once the implementation and the test base class are in place, the tests pass, and both the application and the stub artifacts are built and installed in the local Maven repository. Information about installing the stubs jar to the local repository appears in the logs, as shown in the following example:

[INFO] --- spring-cloud-contract-maven-plugin:1.0.0.BUILD-SNAPSHOT:generateStubs (default-generateStubs) @ http-server ---

[INFO] Building jar: /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-stubs.jar

[INFO]

[INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ http-server ---

[INFO] Building jar: /some/path/http-server/target/http-server-0.0.1-SNAPSHOT.jar

[INFO]

[INFO] --- spring-boot-maven-plugin:1.5.5.BUILD-SNAPSHOT:repackage (default) @ http-server ---

[INFO]

[INFO] --- maven-install-plugin:2.5.2:install (default-install) @ http-server ---

[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT.jar to /path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT.jar

[INFO] Installing /some/path/http-server/pom.xml to /path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT.pom

[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-stubs.jar to /path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar

You can now merge the changes and publish both the application and the stub artifacts in an online repository.

**Docker Project**

In order to enable working with contracts while creating applications in non-JVM technologies, the springcloud/spring-cloud-contract Docker image has been created. It contains a project that automatically generates tests for HTTP contracts and executes them in EXPLICIT test mode. Then, if the tests pass, it generates Wiremock stubs and, optionally, publishes them to an artifact manager. In order to use the image, you can mount the contracts into the /contracts directory and set a few environment variables.

**On the Consumer Side**

Spring Cloud Contract Stub Runner can be used in the integration tests to get a running WireMock instance or messaging route that simulates the actual service.

To get started, add the dependency to Spring Cloud Contract Stub Runner:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-contract-stub-runner</artifactId>

<scope>test</scope>

</dependency>

You can get the Producer-side stubs installed in your Maven repository in either of two ways:

* By checking out the Producer side repository and adding contracts and generating the stubs by running the following commands:
* $ **cd** local-http-server-repo

$ ./mvnw clean install -DskipTests

|  |
| --- |
| [Note] |
| The tests are skipped because the Producer-side contract implementation is not yet in place, so the automatically-generated contract tests fail. |

* Getting already existing producer service stubs from a remote repository. To do so, pass the stub artifact IDs and artifact repository URl as Spring Cloud Contract Stub Runner properties, as shown in the following example:
* stubrunner:
* ids: 'com.example:http-server-dsl:+:stubs:8080'

repositoryRoot: http://repo.spring.io/libs-snapshot

Now you can annotate your test class with @AutoConfigureStubRunner. In the annotation, provide the group-id and artifact-id for Spring Cloud Contract Stub Runner to run the collaborators' stubs for you, as shown in the following example:

*@RunWith(SpringRunner.class)*

*@SpringBootTest(webEnvironment=WebEnvironment.NONE)*

*@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:6565"},*

*stubsMode = StubRunnerProperties.StubsMode.LOCAL)*

*@DirtiesContext*

**public** **class** LoanApplicationServiceTests {

|  |
| --- |
| [Tip] |
| Use the REMOTE stubsMode when downloading stubs from an online repository and LOCAL for offline work. |

In your integration test, you can receive stubbed versions of HTTP responses or messages that are expected to be emitted by the collaborator service. You can see entries similar to the following in the build logs:

2016-07-19 14:22:25.403 INFO 41050 --- [ main] o.s.c.c.stubrunner.AetherStubDownloader : Desired version is + - will try to resolve the latest version

2016-07-19 14:22:25.438 INFO 41050 --- [ main] o.s.c.c.stubrunner.AetherStubDownloader : Resolved version is 0.0.1-SNAPSHOT

2016-07-19 14:22:25.439 INFO 41050 --- [ main] o.s.c.c.stubrunner.AetherStubDownloader : Resolving artifact com.example:http-server:jar:stubs:0.0.1-SNAPSHOT using remote repositories []

2016-07-19 14:22:25.451 INFO 41050 --- [ main] o.s.c.c.stubrunner.AetherStubDownloader : Resolved artifact com.example:http-server:jar:stubs:0.0.1-SNAPSHOT to /path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar

2016-07-19 14:22:25.465 INFO 41050 --- [ main] o.s.c.c.stubrunner.AetherStubDownloader : Unpacking stub from JAR [URI: file:/path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar]

2016-07-19 14:22:25.475 INFO 41050 --- [ main] o.s.c.c.stubrunner.AetherStubDownloader : Unpacked file to [/var/folders/0p/xwq47sq106x1\_g3dtv6qfm940000gq/T/contracts100276532569594265]

2016-07-19 14:22:27.737 INFO 41050 --- [ main] o.s.c.c.stubrunner.StubRunnerExecutor : All stubs are now running RunningStubs [namesAndPorts={com.example:http-server:0.0.1-SNAPSHOT:stubs=8080}]

**89.3.3 Defining the Contract**

As consumers of services, we need to define what exactly we want to achieve. We need to formulate our expectations. That is why we write contracts.

Assume that you want to send a request containing the ID of a client company and the amount it wants to borrow from us. You also want to send it to the /fraudcheck url via the PUT method.

**Groovy DSL.**

**package** contracts

org.springframework.cloud.contract.spec.Contract.make {

request { *// (1)*

method 'PUT' *// (2)*

url '/fraudcheck' *// (3)*

body([ *// (4)*

"client.id": $(regex('[0-9]{10}')),

loanAmount: 99999

])

headers { *// (5)*

contentType('application/json')

}

}

response { *// (6)*

status OK() *// (7)*

body([ *// (8)*

fraudCheckStatus: "FRAUD",

"rejection.reason": "Amount too high"

])

headers { *// (9)*

contentType('application/json')

}

}

}

*/\**

*From the Consumer perspective, when shooting a request in the integration test:*

*(1) - If the consumer sends a request*

*(2) - With the "PUT" method*

*(3) - to the URL "/fraudcheck"*

*(4) - with the JSON body that*

*\* has a field `client.id` that matches a regular expression `[0-9]{10}`*

*\* has a field `loanAmount` that is equal to `99999`*

*(5) - with header `Content-Type` equal to `application/json`*

*(6) - then the response will be sent with*

*(7) - status equal `200`*

*(8) - and JSON body equal to*

*{ "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }*

*(9) - with header `Content-Type` equal to `application/json`*

*From the Producer perspective, in the autogenerated producer-side test:*

*(1) - A request will be sent to the producer*

*(2) - With the "PUT" method*

*(3) - to the URL "/fraudcheck"*

*(4) - with the JSON body that*

*\* has a field `client.id` that will have a generated value that matches a regular expression `[0-9]{10}`*

*\* has a field `loanAmount` that is equal to `99999`*

*(5) - with header `Content-Type` equal to `application/json`*

*(6) - then the test will assert if the response has been sent with*

*(7) - status equal `200`*

*(8) - and JSON body equal to*

*{ "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }*

*(9) - with header `Content-Type` matching `application/json.\*`*

*\*/*

**YAML.**

request: # (1)

method: PUT # (2)

url: /fraudcheck # (3)

body: # (4)

"client.id": 1234567890

loanAmount: 99999

headers: # (5)

Content-Type: application/json

matchers:

body:

- path: $.['client.id'] # (6)

type: by\_regex

value: "[0-9]{10}"

response: # (7)

status: 200 # (8)

body: # (9)

fraudCheckStatus: "FRAUD"

"rejection.reason": "Amount too high"

headers: # (10)

Content-Type: application/json;charset=UTF-8

#From the Consumer perspective, when shooting a request in the integration test:

#

#(1) - If the consumer sends a request

#(2) - With the "PUT" method

#(3) - to the URL "/fraudcheck"

#(4) - with the JSON body that

# \* has a field `client.id`

# \* has a field `loanAmount` that is equal to `99999`

#(5) - with header `Content-Type` equal to `application/json`

#(6) - and a `client.id` json entry matches the regular expression `[0-9]{10}`

#(7) - then the response will be sent with

#(8) - status equal `200`

#(9) - and JSON body equal to

# { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }

#(10) - with header `Content-Type` equal to `application/json`

#

#From the Producer perspective, in the autogenerated producer-side test:

#

#(1) - A request will be sent to the producer

#(2) - With the "PUT" method

#(3) - to the URL "/fraudcheck"

#(4) - with the JSON body that

# \* has a field `client.id` `1234567890`

# \* has a field `loanAmount` that is equal to `99999`

#(5) - with header `Content-Type` equal to `application/json`

#(7) - then the test will assert if the response has been sent with

#(8) - status equal `200`

#(9) - and JSON body equal to

# { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }

#(10) - with header `Content-Type` equal to `application/json;charset=UTF-8`

**89.3.4 Client Side**

Spring Cloud Contract generates stubs, which you can use during client-side testing. You get a running WireMock instance/Messaging route that simulates the service. You would like to feed that instance with a proper stub definition.

At some point in time, you need to send a request to the Fraud Detection service.

ResponseEntity<FraudServiceResponse> response =

restTemplate.exchange("http://localhost:" + port + "/fraudcheck", HttpMethod.PUT,

**new** HttpEntity<>(request, httpHeaders),

FraudServiceResponse.**class**);

Annotate your test class with @AutoConfigureStubRunner. In the annotation provide the group id and artifact id for the Stub Runner to download stubs of your collaborators.

*@RunWith(SpringRunner.class)*

*@SpringBootTest(webEnvironment=WebEnvironment.NONE)*

*@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:6565"},*

*stubsMode = StubRunnerProperties.StubsMode.LOCAL)*

*@DirtiesContext*

**public** **class** LoanApplicationServiceTests {

After that, during the tests, Spring Cloud Contract automatically finds the stubs (simulating the real service) in the Maven repository and exposes them on a configured (or random) port.

**89.3.5 Server Side**

Since you are developing your stub, you need to be sure that it actually resembles your concrete implementation. You cannot have a situation where your stub acts in one way and your application behaves in a different way, especially in production.

To ensure that your application behaves the way you define in your stub, tests are generated from the stub you provide.

The autogenerated test looks, more or less, like this:

*@Test*

**public** **void** validate\_shouldMarkClientAsFraud() **throws** Exception {

*// given:*

MockMvcRequestSpecification request = given()

.header("Content-Type", "application/vnd.fraud.v1+json")

.body("{\"client.id\":\"1234567890\",\"loanAmount\":99999}");

*// when:*

ResponseOptions response = given().spec(request)

.put("/fraudcheck");

*// then:*

assertThat(response.statusCode()).isEqualTo(200);

assertThat(response.header("Content-Type")).matches("application/vnd.fraud.v1.json.\*");

*// and:*

DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());

assertThatJson(parsedJson).field("['fraudCheckStatus']").matches("[A-Z]{5}");

assertThatJson(parsedJson).field("['rejection.reason']").isEqualTo("Amount too high");

}

**89.4 Step-by-step Guide to Consumer Driven Contracts (CDC)**

Consider an example of Fraud Detection and the Loan Issuance process. The business scenario is such that we want to issue loans to people but do not want them to steal from us. The current implementation of our system grants loans to everybody.

Assume that Loan Issuance is a client to the Fraud Detection server. In the current sprint, we must develop a new feature: if a client wants to borrow too much money, then we mark the client as a fraud.

Technical remark - Fraud Detection has an artifact-id of http-server, while Loan Issuance has an artifact-id of http-client, and both have a group-id of com.example.

Social remark - both client and server development teams need to communicate directly and discuss changes while going through the process. CDC is all about communication.

The [server side code is available here](https://github.com/spring-cloud/spring-cloud-contract/tree/master/samples/standalone/dsl/http-server) and [the client code here](https://github.com/spring-cloud/spring-cloud-contract/tree/master/samples/standalone/dsl/http-client).

|  |
| --- |
| [Tip] |
| In this case, the producer owns the contracts. Physically, all the contract are in the producer’s repository. |

**89.4.1 Technical note**

If using the **SNAPSHOT** / **Milestone** / **Release Candidate** versions please add the following section to your build:

**Maven.**

<repositories>

<repository>

<id>spring-snapshots</id>

<name>Spring Snapshots</name>

<url>https://repo.spring.io/snapshot</url>

<snapshots>

<enabled>true</enabled>

</snapshots>

</repository>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

<repository>

<id>spring-releases</id>

<name>Spring Releases</name>

<url>https://repo.spring.io/release</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

<pluginRepositories>

<pluginRepository>

<id>spring-snapshots</id>

<name>Spring Snapshots</name>

<url>https://repo.spring.io/snapshot</url>

<snapshots>

<enabled>true</enabled>

</snapshots>

</pluginRepository>

<pluginRepository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</pluginRepository>

<pluginRepository>

<id>spring-releases</id>

<name>Spring Releases</name>

<url>https://repo.spring.io/release</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</pluginRepository>

</pluginRepositories>

**Gradle.**

repositories {

mavenCentral()

mavenLocal()

maven { url "http://repo.spring.io/snapshot" }

maven { url "http://repo.spring.io/milestone" }

maven { url "http://repo.spring.io/release" }

}

**89.4.2 Consumer side (Loan Issuance)**

As a developer of the Loan Issuance service (a consumer of the Fraud Detection server), you might do the following steps:

1. Start doing TDD by writing a test for your feature.
2. Write the missing implementation.
3. Clone the Fraud Detection service repository locally.
4. Define the contract locally in the repo of Fraud Detection service.
5. Add the Spring Cloud Contract Verifier plugin.
6. Run the integration tests.
7. File a pull request.
8. Create an initial implementation.
9. Take over the pull request.
10. Write the missing implementation.
11. Deploy your app.
12. Work online.

**Start doing TDD by writing a test for your feature.**

*@Test*

**public** **void** shouldBeRejectedDueToAbnormalLoanAmount() {

*// given:*

LoanApplication application = **new** LoanApplication(**new** Client("1234567890"),

99999);

*// when:*

LoanApplicationResult loanApplication = service.loanApplication(application);

*// then:*

assertThat(loanApplication.getLoanApplicationStatus())

.isEqualTo(LoanApplicationStatus.LOAN\_APPLICATION\_REJECTED);

assertThat(loanApplication.getRejectionReason()).isEqualTo("Amount too high");

}

Assume that you have written a test of your new feature. If a loan application for a big amount is received, the system should reject that loan application with some description.

**Write the missing implementation.**

At some point in time, you need to send a request to the Fraud Detection service. Assume that you need to send the request containing the ID of the client and the amount the client wants to borrow. You want to send it to the /fraudcheck url via the PUT method.

ResponseEntity<FraudServiceResponse> response =

restTemplate.exchange("http://localhost:" + port + "/fraudcheck", HttpMethod.PUT,

**new** HttpEntity<>(request, httpHeaders),

FraudServiceResponse.**class**);

For simplicity, the port of the Fraud Detection service is set to 8080, and the application runs on 8090.

If you start the test at this point, it breaks, because no service currently runs on port 8080.

**Clone the Fraud Detection service repository locally.**

You can start by playing around with the server side contract. To do so, you must first clone it.

$ git clone https://your-git-server.com/server-side.git local-http-server-repo

**Define the contract locally in the repo of Fraud Detection service.**

As a consumer, you need to define what exactly you want to achieve. You need to formulate your expectations. To do so, write the following contract:

|  |  |
| --- | --- |
| [Important] | **Important** |
| Place the contract under src/test/resources/contracts/fraud folder. The fraud folder is important because the producer’s test base class name references that folder. |

**Groovy DSL.**

**package** contracts

org.springframework.cloud.contract.spec.Contract.make {

request { *// (1)*

method 'PUT' *// (2)*

url '/fraudcheck' *// (3)*

body([ *// (4)*

"client.id": $(regex('[0-9]{10}')),

loanAmount: 99999

])

headers { *// (5)*

contentType('application/json')

}

}

response { *// (6)*

status OK() *// (7)*

body([ *// (8)*

fraudCheckStatus: "FRAUD",

"rejection.reason": "Amount too high"

])

headers { *// (9)*

contentType('application/json')

}

}

}

*/\**

*From the Consumer perspective, when shooting a request in the integration test:*

*(1) - If the consumer sends a request*

*(2) - With the "PUT" method*

*(3) - to the URL "/fraudcheck"*

*(4) - with the JSON body that*

*\* has a field `client.id` that matches a regular expression `[0-9]{10}`*

*\* has a field `loanAmount` that is equal to `99999`*

*(5) - with header `Content-Type` equal to `application/json`*

*(6) - then the response will be sent with*

*(7) - status equal `200`*

*(8) - and JSON body equal to*

*{ "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }*

*(9) - with header `Content-Type` equal to `application/json`*

*From the Producer perspective, in the autogenerated producer-side test:*

*(1) - A request will be sent to the producer*

*(2) - With the "PUT" method*

*(3) - to the URL "/fraudcheck"*

*(4) - with the JSON body that*

*\* has a field `client.id` that will have a generated value that matches a regular expression `[0-9]{10}`*

*\* has a field `loanAmount` that is equal to `99999`*

*(5) - with header `Content-Type` equal to `application/json`*

*(6) - then the test will assert if the response has been sent with*

*(7) - status equal `200`*

*(8) - and JSON body equal to*

*{ "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }*

*(9) - with header `Content-Type` matching `application/json.\*`*

*\*/*

**YAML.**

request: # (1)

method: PUT # (2)

url: /fraudcheck # (3)

body: # (4)

"client.id": 1234567890

loanAmount: 99999

headers: # (5)

Content-Type: application/json

matchers:

body:

- path: $.['client.id'] # (6)

type: by\_regex

value: "[0-9]{10}"

response: # (7)

status: 200 # (8)

body: # (9)

fraudCheckStatus: "FRAUD"

"rejection.reason": "Amount too high"

headers: # (10)

Content-Type: application/json;charset=UTF-8

#From the Consumer perspective, when shooting a request in the integration test:

#

#(1) - If the consumer sends a request

#(2) - With the "PUT" method

#(3) - to the URL "/fraudcheck"

#(4) - with the JSON body that

# \* has a field `client.id`

# \* has a field `loanAmount` that is equal to `99999`

#(5) - with header `Content-Type` equal to `application/json`

#(6) - and a `client.id` json entry matches the regular expression `[0-9]{10}`

#(7) - then the response will be sent with

#(8) - status equal `200`

#(9) - and JSON body equal to

# { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }

#(10) - with header `Content-Type` equal to `application/json`

#

#From the Producer perspective, in the autogenerated producer-side test:

#

#(1) - A request will be sent to the producer

#(2) - With the "PUT" method

#(3) - to the URL "/fraudcheck"

#(4) - with the JSON body that

# \* has a field `client.id` `1234567890`

# \* has a field `loanAmount` that is equal to `99999`

#(5) - with header `Content-Type` equal to `application/json`

#(7) - then the test will assert if the response has been sent with

#(8) - status equal `200`

#(9) - and JSON body equal to

# { "fraudCheckStatus": "FRAUD", "rejectionReason": "Amount too high" }

#(10) - with header `Content-Type` equal to `application/json;charset=UTF-8`

The YML contract is quite straight-forward. However when you take a look at the Contract written using a statically typed Groovy DSL - you might wonder what thevalue(client(…​), server(…​)) parts are. By using this notation, Spring Cloud Contract lets you define parts of a JSON block, a URL, etc., which are dynamic. In case of an identifier or a timestamp, you need not hardcode a value. You want to allow some different ranges of values. To enable ranges of values, you can set regular expressions matching those values for the consumer side. You can provide the body by means of either a map notation or String with interpolations. Consult the [???](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html)section for more information. We highly recommend using the map notation!

|  |
| --- |
| [Tip] |
| You must understand the map notation in order to set up contracts. Please read the [Groovy docs regarding JSON](http://groovy-lang.org/json.html). |

The previously shown contract is an agreement between two sides that:

* if an HTTP request is sent with all of
  + a PUT method on the /fraudcheck endpoint,
  + a JSON body with a client.id that matches the regular expression [0-9]{10} and loanAmount equal to 99999,
  + and a Content-Type header with a value of application/vnd.fraud.v1+json,
* then an HTTP response is sent to the consumer that
  + has status 200,
  + contains a JSON body with the fraudCheckStatus field containing a value FRAUD and the rejectionReason field having value Amount too high,
  + and a Content-Type header with a value of application/vnd.fraud.v1+json.

Once you are ready to check the API in practice in the integration tests, you need to install the stubs locally.

**Add the Spring Cloud Contract Verifier plugin.**

We can add either a Maven or a Gradle plugin. In this example, you see how to add Maven. First, add the Spring Cloud Contract BOM.

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>${spring-cloud-dependencies.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

Next, add the Spring Cloud Contract Verifier Maven plugin

<plugin>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions>

<configuration>

<packageWithBaseClasses>com.example.fraud</packageWithBaseClasses>

</configuration>

</plugin>

Since the plugin was added, you get the Spring Cloud Contract Verifier features which, from the provided contracts:

* generate and run tests
* produce and install stubs

You do not want to generate tests since you, as the consumer, want only to play with the stubs. You need to skip the test generation and execution. When you execute:

$ **cd** local-http-server-repo

$ ./mvnw clean install -DskipTests

In the logs, you see something like this:

[INFO] --- spring-cloud-contract-maven-plugin:1.0.0.BUILD-SNAPSHOT:generateStubs (default-generateStubs) @ http-server ---

[INFO] Building jar: /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-stubs.jar

[INFO]

[INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ http-server ---

[INFO] Building jar: /some/path/http-server/target/http-server-0.0.1-SNAPSHOT.jar

[INFO]

[INFO] --- spring-boot-maven-plugin:1.5.5.BUILD-SNAPSHOT:repackage (default) @ http-server ---

[INFO]

[INFO] --- maven-install-plugin:2.5.2:install (default-install) @ http-server ---

[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT.jar to /path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT.jar

[INFO] Installing /some/path/http-server/pom.xml to /path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT.pom

[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-stubs.jar to /path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar

The following line is extremely important:

[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-stubs.jar to /path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar

It confirms that the stubs of the http-server have been installed in the local repository.

**Run the integration tests.**

In order to profit from the Spring Cloud Contract Stub Runner functionality of automatic stub downloading, you must do the following in your consumer side project (Loan Application service):

Add the Spring Cloud Contract BOM:

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>${spring-cloud-dependencies.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

Add the dependency to Spring Cloud Contract Stub Runner:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-contract-stub-runner</artifactId>

<scope>test</scope>

</dependency>

Annotate your test class with @AutoConfigureStubRunner. In the annotation, provide the group-id and artifact-id for the Stub Runner to download the stubs of your collaborators. (Optional step) Because you’re playing with the collaborators offline, you can also provide the offline work switch (StubRunnerProperties.StubsMode.LOCAL).

*@RunWith(SpringRunner.class)*

*@SpringBootTest(webEnvironment=WebEnvironment.NONE)*

*@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:6565"},*

*stubsMode = StubRunnerProperties.StubsMode.LOCAL)*

*@DirtiesContext*

**public** **class** LoanApplicationServiceTests {

Now, when you run your tests, you see something like this:

2016-07-19 14:22:25.403 INFO 41050 --- [ main] o.s.c.c.stubrunner.AetherStubDownloader : Desired version is + - will try to resolve the latest version

2016-07-19 14:22:25.438 INFO 41050 --- [ main] o.s.c.c.stubrunner.AetherStubDownloader : Resolved version is 0.0.1-SNAPSHOT

2016-07-19 14:22:25.439 INFO 41050 --- [ main] o.s.c.c.stubrunner.AetherStubDownloader : Resolving artifact com.example:http-server:jar:stubs:0.0.1-SNAPSHOT using remote repositories []

2016-07-19 14:22:25.451 INFO 41050 --- [ main] o.s.c.c.stubrunner.AetherStubDownloader : Resolved artifact com.example:http-server:jar:stubs:0.0.1-SNAPSHOT to /path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar

2016-07-19 14:22:25.465 INFO 41050 --- [ main] o.s.c.c.stubrunner.AetherStubDownloader : Unpacking stub from JAR [URI: file:/path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar]

2016-07-19 14:22:25.475 INFO 41050 --- [ main] o.s.c.c.stubrunner.AetherStubDownloader : Unpacked file to [/var/folders/0p/xwq47sq106x1\_g3dtv6qfm940000gq/T/contracts100276532569594265]

2016-07-19 14:22:27.737 INFO 41050 --- [ main] o.s.c.c.stubrunner.StubRunnerExecutor : All stubs are now running RunningStubs [namesAndPorts={com.example:http-server:0.0.1-SNAPSHOT:stubs=8080}]

This output means that Stub Runner has found your stubs and started a server for your app with group id com.example, artifact id http-server with version 0.0.1-SNAPSHOT of the stubs and with stubs classifier on port 8080.

**File a pull request.**

What you have done until now is an iterative process. You can play around with the contract, install it locally, and work on the consumer side until the contract works as you wish.

Once you are satisfied with the results and the test passes, publish a pull request to the server side. Currently, the consumer side work is done.

**89.4.3 Producer side (Fraud Detection server)**

As a developer of the Fraud Detection server (a server to the Loan Issuance service):

**Create an initial implementation.**

As a reminder, you can see the initial implementation here:

*@RequestMapping(value = "/fraudcheck", method = PUT)*

**public** FraudCheckResult fraudCheck(*@RequestBody* FraudCheck fraudCheck) {

**return** **new** FraudCheckResult(FraudCheckStatus.OK, NO\_REASON);

}

**Take over the pull request.**

$ git checkout -b contract-change-pr master

$ git pull https://your-git-server.com/server-side-fork.git contract-change-pr

You must add the dependencies needed by the autogenerated tests:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-contract-verifier</artifactId>

<scope>test</scope>

</dependency>

In the configuration of the Maven plugin, pass the packageWithBaseClasses property

<plugin>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions>

<configuration>

<packageWithBaseClasses>com.example.fraud</packageWithBaseClasses>

</configuration>

</plugin>

|  |  |
| --- | --- |
| [Important] | **Important** |
| This example uses "convention based" naming by setting the packageWithBaseClasses property. Doing so means that the two last packages combine to make the name of the base test class. In our case, the contracts were placed under src/test/resources/contracts/fraud. Since you do not have two packages starting from the contracts folder, pick only one, which should be fraud. Add the Base suffix and capitalize fraud. That gives you the FraudBase test class name. |

All the generated tests extend that class. Over there, you can set up your Spring Context or whatever is necessary. In this case, use [Rest Assured MVC](http://rest-assured.io/) to start the server side FraudDetectionController.

**package** com.example.fraud;

**import** org.junit.Before;

**import** io.restassured.module.mockmvc.RestAssuredMockMvc;

**public** **class** FraudBase {

*@Before*

**public** **void** setup() {

RestAssuredMockMvc.standaloneSetup(**new** FraudDetectionController(),

**new** FraudStatsController(stubbedStatsProvider()));

}

**private** StatsProvider stubbedStatsProvider() {

**return** fraudType -> {

**switch** (fraudType) {

**case** DRUNKS:

**return** 100;

**case** ALL:

**return** 200;

}

**return** 0;

};

}

**public** **void** assertThatRejectionReasonIsNull(Object rejectionReason) {

assert rejectionReason == null;

}

}

Now, if you run the ./mvnw clean install, you get something like this:

Results :

Tests in error:

ContractVerifierTest.validate\_shouldMarkClientAsFraud:32 » IllegalState Parsed...

This error occurs because you have a new contract from which a test was generated and it failed since you have not implemented the feature. The auto-generated test would look like this:

*@Test*

**public** **void** validate\_shouldMarkClientAsFraud() **throws** Exception {

*// given:*

MockMvcRequestSpecification request = given()

.header("Content-Type", "application/vnd.fraud.v1+json")

.body("{\"client.id\":\"1234567890\",\"loanAmount\":99999}");

*// when:*

ResponseOptions response = given().spec(request)

.put("/fraudcheck");

*// then:*

assertThat(response.statusCode()).isEqualTo(200);

assertThat(response.header("Content-Type")).matches("application/vnd.fraud.v1.json.\*");

*// and:*

DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());

assertThatJson(parsedJson).field("['fraudCheckStatus']").matches("[A-Z]{5}");

assertThatJson(parsedJson).field("['rejection.reason']").isEqualTo("Amount too high");

}

If you used the Groovy DSL, you can see, all the producer() parts of the Contract that were present in the value(consumer(…​), producer(…​)) blocks got injected into the test. In case of using YAML, the same applied for the matchers sections of the response.

Note that, on the producer side, you are also doing TDD. The expectations are expressed in the form of a test. This test sends a request to our own application with the URL, headers, and body defined in the contract. It also is expecting precisely defined values in the response. In other words, you have the red part of red, green, andrefactor. It is time to convert the red into the green.

**Write the missing implementation.**

Because you know the expected input and expected output, you can write the missing implementation:

*@RequestMapping(value = "/fraudcheck", method = PUT)*

**public** FraudCheckResult fraudCheck(*@RequestBody* FraudCheck fraudCheck) {

**if** (amountGreaterThanThreshold(fraudCheck)) {

**return** **new** FraudCheckResult(FraudCheckStatus.FRAUD, AMOUNT\_TOO\_HIGH);

}

**return** **new** FraudCheckResult(FraudCheckStatus.OK, NO\_REASON);

}

When you execute ./mvnw clean install again, the tests pass. Since the Spring Cloud Contract Verifier plugin adds the tests to the generated-test-sources, you can actually run those tests from your IDE.

**Deploy your app.**

Once you finish your work, you can deploy your change. First, merge the branch:

$ git checkout master

$ git merge --no-ff contract-change-pr

$ git push origin master

Your CI might run something like ./mvnw clean deploy, which would publish both the application and the stub artifacts.

**89.4.4 Consumer Side (Loan Issuance) Final Step**

As a developer of the Loan Issuance service (a consumer of the Fraud Detection server):

**Merge branch to master.**

$ git checkout master

$ git merge --no-ff contract-change-pr

**Work online.**

Now you can disable the offline work for Spring Cloud Contract Stub Runner and indicate where the repository with your stubs is located. At this moment the stubs of the server side are automatically downloaded from Nexus/Artifactory. You can set the value of stubsMode to REMOTE. The following code shows an example of achieving the same thing by changing the properties.

stubrunner:

ids: 'com.example:http-server-dsl:+:stubs:8080'

repositoryRoot: http://repo.spring.io/libs-snapshot

That’s it!

**89.5 Dependencies**

The best way to add dependencies is to use the proper starter dependency.

For stub-runner, use spring-cloud-starter-stub-runner. When you use a plugin, add spring-cloud-starter-contract-verifier.

**89.6 Additional Links**

Here are some resources related to Spring Cloud Contract Verifier and Stub Runner. Note that some may be outdated, because the Spring Cloud Contract Verifier project is under constant development.

**89.6.1 Spring Cloud Contract video**

You can check out the video from the Warsaw JUG about Spring Cloud Contract:

**89.6.2 Readings**

* [Slides from Marcin Grzejszczak’s talk about Accurest](http://www.slideshare.net/MarcinGrzejszczak/stick-to-the-rules-consumer-driven-contracts-201507-confitura)
* [Accurest related articles from Marcin Grzejszczak’s blog](http://toomuchcoding.com/blog/categories/accurest/)
* [Spring Cloud Contract related articles from Marcin Grzejszczak’s blog](http://toomuchcoding.com/blog/categories/spring-cloud-contract/)
* [Groovy docs regarding JSON](http://groovy-lang.org/json.html)

**89.7 Samples**

You can find some samples at [samples](https://github.com/spring-cloud-samples/spring-cloud-contract-samples).

**90. Spring Cloud Contract FAQ**

**90.1 Why use Spring Cloud Contract Verifier and not X ?**

For the time being Spring Cloud Contract is a JVM based tool. So it could be your first pick when you’re already creating software for the JVM. This project has a lot of really interesting features but especially quite a few of them definitely make Spring Cloud Contract Verifier stand out on the "market" of Consumer Driven Contract (CDC) tooling. Out of many the most interesting are:

* Possibility to do CDC with messaging
* Clear and easy to use, statically typed DSL
* Possibility to copy paste your current JSON file to the contract and only edit its elements
* Automatic generation of tests from the defined Contract
* Stub Runner functionality - the stubs are automatically downloaded at runtime from Nexus / Artifactory
* Spring Cloud integration - no discovery service is needed for integration tests
* Spring Cloud Contract integrates with Pact out of the box and provides easy hooks to extend its functionality
* Via Docker adds support for any language & framework used

**90.2 I don’t want to write a contract in Groovy!**

No problem. You can write a contract in YAML!

**90.3 What is this value(consumer(), producer()) ?**

One of the biggest challenges related to stubs is their reusability. Only if they can be vastly used, will they serve their purpose. What typically makes that difficult are the hard-coded values of request / response elements. For example dates or ids. Imagine the following JSON request

**{**

"time" : "2016-10-10 20:10:15"**,**

"id" : "9febab1c-6f36-4a0b-88d6-3b6a6d81cd4a"**,**

"body" : "foo"

**}**

and JSON response

**{**

"time" : "2016-10-10 21:10:15"**,**

"id" : "c4231e1f-3ca9-48d3-b7e7-567d55f0d051"**,**

"body" : "bar"

**}**

Imagine the pain required to set proper value of the time field (let’s assume that this content is generated by the database) by changing the clock in the system or providing stub implementations of data providers. The same is related to the field called id. Will you create a stubbed implementation of UUID generator? Makes little sense…​

So as a consumer you would like to send a request that matches any form of a time or any UUID. That way your system will work as usual - will generate data and you won’t have to stub anything out. Let’s assume that in case of the aforementioned JSON the most important part is the body field. You can focus on that and provide matching for other fields. In other words you would like the stub to work like this:

**{**

"time" : "SOMETHING THAT MATCHES TIME"**,**

"id" : "SOMETHING THAT MATCHES UUID"**,**

"body" : "foo"

**}**

As far as the response goes as a consumer you need a concrete value that you can operate on. So such a JSON is valid

**{**

"time" : "2016-10-10 21:10:15"**,**

"id" : "c4231e1f-3ca9-48d3-b7e7-567d55f0d051"**,**

"body" : "bar"

**}**

As you could see in the previous sections we generate tests from contracts. So from the producer’s side the situation looks much different. We’re parsing the provided contract and in the test we want to send a real request to your endpoints. So for the case of a producer for the request we can’t have any sort of matching. We need concrete values that the producer’s backend can work on. Such a JSON would be a valid one:

**{**

"time" : "2016-10-10 20:10:15"**,**

"id" : "9febab1c-6f36-4a0b-88d6-3b6a6d81cd4a"**,**

"body" : "foo"

**}**

On the other hand from the point of view of the validity of the contract the response doesn’t necessarily have to contain concrete values of time or id. Let’s say that you generate those on the producer side - again, you’d have to do a lot of stubbing to ensure that you always return the same values. That’s why from the producer’s side what you might want is the following response:

**{**

"time" : "SOMETHING THAT MATCHES TIME"**,**

"id" : "SOMETHING THAT MATCHES UUID"**,**

"body" : "bar"

**}**

How can you then provide one time a matcher for the consumer and a concrete value for the producer and vice versa? In Spring Cloud Contract we’re allowing you to provide a **dynamic value**. That means that it can differ for both sides of the communication. You can pass the values:

Either via the value method

value(consumer(...), producer(...))

value(stub(...), test(...))

value(client(...), server(...))

or using the $() method

$(consumer(...), producer(...))

$(stub(...), test(...))

$(client(...), server(...))

You can read more about this in the [???](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html) section.

Calling value() or $() tells Spring Cloud Contract that you will be passing a dynamic value. Inside the consumer() method you pass the value that should be used on the consumer side (in the generated stub). Inside the producer() method you pass the value that should be used on the producer side (in the generated test).

|  |
| --- |
| [Tip] |
| If on one side you have passed the regular expression and you haven’t passed the other, then the other side will get auto-generated. |

Most often you will use that method together with the regex helper method. E.g. consumer(regex('[0-9]{10}')).

To sum it up the contract for the aforementioned scenario would look more or less like this (the regular expression for time and UUID are simplified and most likely invalid but we want to keep things very simple in this example):

org.springframework.cloud.contract.spec.Contract.make {

request {

method 'GET'

url '/someUrl'

body([

time : value(consumer(regex('[0-9]{4}-[0-9]{2}-[0-9]{2} [0-2][0-9]-[0-5][0-9]-[0-5][0-9]')),

id: value(consumer(regex('[0-9a-zA-z]{8}-[0-9a-zA-z]{4}-[0-9a-zA-z]{4}-[0-9a-zA-z]{12}'))

body: "foo"

])

}

response {

status OK()

body([

time : value(producer(regex('[0-9]{4}-[0-9]{2}-[0-9]{2} [0-2][0-9]-[0-5][0-9]-[0-5][0-9]')),

id: value([producer(regex('[0-9a-zA-z]{8}-[0-9a-zA-z]{4}-[0-9a-zA-z]{4}-[0-9a-zA-z]{12}'))

body: "bar"

])

}

}

|  |  |
| --- | --- |
| [Important] | **Important** |
| Please read the [Groovy docs related to JSON](http://groovy-lang.org/json.html) to understand how to properly structure the request / response bodies. |

**90.4 How to do Stubs versioning?**

**90.4.1 API Versioning**

Let’s try to answer a question what versioning really means. If you’re referring to the API version then there are different approaches.

* use Hypermedia, links and do not version your API by any means
* pass versions through headers / urls

I will not try to answer a question which approach is better. Whatever suit your needs and allows you to generate business value should be picked.

Let’s assume that you do version your API. In that case you should provide as many contracts as many versions you support. You can create a subfolder for every version or append it to th contract name - whatever suits you more.

**90.4.2 JAR versioning**

If by versioning you mean the version of the JAR that contains the stubs then there are essentially two main approaches.

Let’s assume that you’re doing Continuous Delivery / Deployment which means that you’re generating a new version of the jar each time you go through the pipeline and that jar can go to production at any time. For example your jar version looks like this (it got built on the 20.10.2016 at 20:15:21) :

1.0.0.20161020-201521-RELEASE

In that case your generated stub jar will look like this.

1.0.0.20161020-201521-RELEASE-stubs.jar

In this case you should inside your application.yml or @AutoConfigureStubRunner when referencing stubs provide the latest version of the stubs. You can do that by passing the + sign. Example

@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:8080"})

If the versioning however is fixed (e.g. 1.0.4.RELEASE or 2.1.1) then you have to set the concrete value of the jar version. Example for 2.1.1.

@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:2.1.1:stubs:8080"})

**90.4.3 Dev or prod stubs**

You can manipulate the classifier to run the tests against current development version of the stubs of other services or the ones that were deployed to production. If you alter your build to deploy the stubs with the prod-stubs classifier once you reach production deployment then you can run tests in one case with dev stubs and one with prod stubs.

Example of tests using development version of stubs

@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:8080"})

Example of tests using production version of stubs

@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:prod-stubs:8080"})

You can pass those values also via properties from your deployment pipeline.

**90.5 Common repo with contracts**

Another way of storing contracts other than having them with the producer is keeping them in a common place. It can be related to security issues where the consumers can’t clone the producer’s code. Also if you keep contracts in a single place then you, as a producer, will know how many consumers you have and which consumer will you break with your local changes.

**90.5.1 Repo structure**

Let’s assume that we have a producer with coordinates com.example:server and 3 consumers: client1, client2, client3. Then in the repository with common contracts you would have the following setup (which you can checkout [here](https://github.com/spring-cloud/spring-cloud-contract/tree/master/samples/standalone/contracts)):

├── com

│   └── example

│   └── server

│   ├── client1

│   │   └── expectation.groovy

│   ├── client2

│   │   └── expectation.groovy

│   ├── client3

│   │   └── expectation.groovy

│   └── pom.xml

├── mvnw

├── mvnw.cmd

├── pom.xml

└── src

└── assembly

└── contracts.xml

As you can see the under the slash-delimited groupid / artifact id folder (com/example/server) you have expectations of the 3 consumers (client1, client2 and client3). Expectations are the standard Groovy DSL contract files as described throughout this documentation. This repository has to produce a JAR file that maps one to one to the contents of the repo.

Example of a pom.xml inside the server folder.

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>server</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>Server Stubs</name>

<description>POM used to install locally stubs for consumer side</description>

<parent>

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

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

<version>2.0.3.RELEASE</version>

<relativePath />

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<java.version>1.8</java.version>

<spring-cloud-contract.version>2.1.0.BUILD-SNAPSHOT</spring-cloud-contract.version>

<spring-cloud-dependencies.version>Finchley.BUILD-SNAPSHOT</spring-cloud-dependencies.version>

<excludeBuildFolders>true</excludeBuildFolders>

</properties>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>${spring-cloud-dependencies.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<build>

<plugins>

<plugin>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions>

<configuration>

*<!-- By default it would search under src/test/resources/ -->*

<contractsDirectory>${project.basedir}</contractsDirectory>

</configuration>

</plugin>

</plugins>

</build>

<repositories>

<repository>

<id>spring-snapshots</id>

<name>Spring Snapshots</name>

<url>https://repo.spring.io/snapshot</url>

<snapshots>

<enabled>true</enabled>

</snapshots>

</repository>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

<repository>

<id>spring-releases</id>

<name>Spring Releases</name>

<url>https://repo.spring.io/release</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

<pluginRepositories>

<pluginRepository>

<id>spring-snapshots</id>

<name>Spring Snapshots</name>

<url>https://repo.spring.io/snapshot</url>

<snapshots>

<enabled>true</enabled>

</snapshots>

</pluginRepository>

<pluginRepository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</pluginRepository>

<pluginRepository>

<id>spring-releases</id>

<name>Spring Releases</name>

<url>https://repo.spring.io/release</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</pluginRepository>

</pluginRepositories>

</project>

As you can see there are no dependencies other than the Spring Cloud Contract Maven Plugin. Those poms are necessary for the consumer side to run mvn clean install -DskipTests to locally install stubs of the producer project.

The pom.xml in the root folder can look like this:

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.example.standalone</groupId>

<artifactId>contracts</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>Contracts</name>

<description>Contains all the Spring Cloud Contracts, well, contracts. JAR used by the producers to generate tests and stubs</description>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-assembly-plugin</artifactId>

<executions>

<execution>

<id>contracts</id>

<phase>prepare-package</phase>

<goals>

<goal>single</goal>

</goals>

<configuration>

<attach>true</attach>

<descriptor>${basedir}/src/assembly/contracts.xml</descriptor>

*<!-- If you want an explicit classifier remove the following line -->*

<appendAssemblyId>false</appendAssemblyId>

</configuration>

</execution>

</executions>

</plugin>

</plugins>

</build>

</project>

It’s using the assembly plugin in order to build the JAR with all the contracts. Example of such setup is here:

<assembly xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3 http://maven.apache.org/xsd/assembly-1.1.3.xsd">

<id>project</id>

<formats>

<format>jar</format>

</formats>

<includeBaseDirectory>false</includeBaseDirectory>

<fileSets>

<fileSet>

<directory>${project.basedir}</directory>

<outputDirectory>/</outputDirectory>

<useDefaultExcludes>true</useDefaultExcludes>

<excludes>

<exclude>\*\*/${project.build.directory}/\*\*</exclude>

<exclude>mvnw</exclude>

<exclude>mvnw.cmd</exclude>

<exclude>.mvn/\*\*</exclude>

<exclude>src/\*\*</exclude>

</excludes>

</fileSet>

</fileSets>

</assembly>

**90.5.2 Workflow**

The workflow would look similar to the one presented in the Step by step guide to CDC. The only difference is that the producer doesn’t own the contracts anymore. So the consumer and the producer have to work on common contracts in a common repository.

**90.5.3 Consumer**

When the **consumer** wants to work on the contracts offline, instead of cloning the producer code, the consumer team clones the common repository, goes to the required producer’s folder (e.g. com/example/server) and runs mvn clean install -DskipTests to install locally the stubs converted from the contracts.

|  |
| --- |
| [Tip] |
| You need to have [Maven installed locally](https://maven.apache.org/download.cgi) |

**90.5.4 Producer**

As a **producer** it’s enough to alter the Spring Cloud Contract Verifier to provide the URL and the dependency of the JAR containing the contracts:

<plugin>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<configuration>

<contractsMode>REMOTE</contractsMode>

<contractsRepositoryUrl>http://link/to/your/nexus/or/artifactory/or/sth</contractsRepositoryUrl>

<contractDependency>

<groupId>com.example.standalone</groupId>

<artifactId>contracts</artifactId>

</contractDependency>

</configuration>

</plugin>

With this setup the JAR with groupid com.example.standalone and artifactid contracts will be downloaded from <http://link/to/your/nexus/or/artifactory/or/sth>. It will be then unpacked in a local temporary folder and contracts present under the com/example/serverwill be picked as the ones used to generate the tests and the stubs. Due to this convention the producer team will know which consumer teams will be broken when some incompatible changes are done.

The rest of the flow looks the same.

**90.5.5 How can I define messaging contracts per topic not per producer?**

To avoid messaging contracts duplication in the common repo, when few producers writing messages to one topic, we could create the structure when the rest contracts would be placed in a folder per producer and messaging contracts in the folder per topic.

**For Maven Project**

To make it possible to work on the producer side we could do the following things (all via Maven plugins):

* Add common repo dependency to your classpath:

<dependency>

<groupId>com.example</groupId>

<artifactId>common-repo</artifactId>

<version>${common-repo.version}</version>

</dependency>

* Download the JAR with the contracts and unpack the JAR to target:

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-dependency-plugin</artifactId>

<version>3.0.0</version>

<executions>

<execution>

<id>unpack-dependencies</id>

<phase>process-resources</phase>

<goals>

<goal>unpack</goal>

</goals>

<configuration>

<artifactItems>

<artifactItem>

<groupId>com.example</groupId>

<artifactId>common-repo</artifactId>

<type>jar</type>

<overWrite>false</overWrite>

<outputDirectory>${project.build.directory}/contracts</outputDirectory>

</artifactItem>

</artifactItems>

</configuration>

</execution>

</executions>

</plugin>

* Rip out all the folders we’re not interested in:

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-antrun-plugin</artifactId>

<version>1.8</version>

<executions>

<execution>

<phase>process-resources</phase>

<goals>

<goal>run</goal>

</goals>

<configuration>

<tasks>

<delete includeemptydirs="true">

<fileset dir="${project.build.directory}/contracts">

<include name="\*\*/\*" />

*<!--Producer artifactId-->*

<exclude name="\*\*/${project.artifactId}/\*\*" />

*<!--List of the supported topics-->*

<exclude name="\*\*/${first-topic}/\*\*" />

<exclude name="\*\*/${second-topic}/\*\*" />

</fileset>

</delete>

</tasks>

</configuration>

</execution>

</executions>

</plugin>

* Run the contract plugin by pointing to the contracts to the folder under target:

<plugin>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions>

<configuration>

<packageWithBaseClasses>com.example</packageWithBaseClasses>

<baseClassMappings>

<baseClassMapping>

<contractPackageRegex>.\*intoxication.\*</contractPackageRegex>

<baseClassFQN>com.example.intoxication.BeerIntoxicationBase</baseClassFQN>

</baseClassMapping>

</baseClassMappings>

<contractsDirectory>${project.build.directory}/contracts</contractsDirectory>

</configuration>

</plugin>

**For Gradle Project**

* Add a custom configuration for the common-repo dependency:

ext {

conractsGroupId = "com.example"

contractsArtifactId = "common-repo"

contractsVersion = "1.2.3"

}

configurations {

contracts {

transitive = false

}

}

* Add the common-repo dependency to your classpath:

dependencies {

contracts "${conractsGroupId}:${contractsArtifactId}:${contractsVersion}"

testCompile "${conractsGroupId}:${contractsArtifactId}:${contractsVersion}"

}

* Download the dependency to an appropriate folder:

task getContracts(type: Copy) {

from configurations.contracts

into **new** File(project.buildDir, "downloadedContracts")

}

* Unzip JAR:

task unzipContracts(type: Copy) {

def zipFile = **new** File(project.buildDir, "downloadedContracts/${contractsArtifactId}-${contractsVersion}.jar")

def outputDir = file("${buildDir}/unpackedContracts")

from zipTree(zipFile)

into outputDir

}

* Cleanup unused contracts:

task deleteUnwantedContracts(type: Delete) {

delete fileTree(dir: "${buildDir}/unpackedContracts",

include: "\*\*/\*",

excludes: [

"\*\*/${project.name}/\*\*"",

"\*\*/${first-topic}/\*\*",

"\*\*/${second-topic}/\*\*"])

}

* Create task dependencies:

unzipContracts.dependsOn("getContracts")

deleteUnwantedContracts.dependsOn("unzipContracts")

build.dependsOn("deleteUnwantedContracts")

* Configure plugin by specifying the directory containing contracts using contractsDslDir property

contracts {

contractsDslDir = **new** File("${buildDir}/unpackedContracts")

}

**90.6 Do I need a Binary Storage? Can’t I use Git?**

In the polyglot world, there are languages that don’t use binary storages like Artifactory or Nexus. Starting from Spring Cloud Contract version 2.0.0 we provide mechanisms to store contracts and stubs in a SCM repository. Currently the only supported SCM is Git.

The repository would have to the following setup (which you can checkout [here](https://github.com/spring-cloud-samples/spring-cloud-contract-samples/tree/2.0.x/contracts_git/)):

.

└── META-INF

└── com.example

└── beer-api-producer-git

└── 0.0.1-SNAPSHOT

├── contracts

│   └── beer-api-consumer

│   ├── messaging

│   │   ├── shouldSendAcceptedVerification.groovy

│   │   └── shouldSendRejectedVerification.groovy

│   └── rest

│   ├── shouldGrantABeerIfOldEnough.groovy

│   └── shouldRejectABeerIfTooYoung.groovy

└── mappings

└── beer-api-consumer

└── rest

├── shouldGrantABeerIfOldEnough.json

└── shouldRejectABeerIfTooYoung.json

Under META-INF folder:

* we group applications via groupId (e.g. com.example)
* then each application is represented via the artifactId (e.g. beer-api-producer-git)
* next, the version of the application. The version is mandatory! (e.g. 0.0.1-SNAPSHOT)
* finally, there are two folders:
  + contracts - the good practice is to store the contracts required by each consumer in the folder with the consumer name (e.g. beer-api-consumer). That way you can use the stubs-per-consumer feature. Further directory structure is arbitrary.
  + mappings - in this folder the Maven / Gradle Spring Cloud Contract plugins will push the stub server mappings. On the consumer side, Stub Runner will scan this folder to start stub servers with stub definitions. The folder structure will be a copy of the one created in the contracts subfolder.

**90.6.1 Protocol convention**

In order to control the type and location of the source of contracts (whether it’s a binary storage or an SCM repository), you can use the protocol in the URL of the repository. Spring Cloud Contract iterates over registered protocol resolvers and tries to fetch the contracts (via a plugin) or stubs (via Stub Runner).

For the SCM functionality, currently, we support the Git repository. To use it, in the property, where the repository URL needs to be placed you just have to prefix the connection URL with git://. Here you can find a couple of examples:

git://file:///foo/bar

git://https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs-contracts-git.git

git://git@github.com:spring-cloud-samples/spring-cloud-contract-nodejs-contracts-git.git

**90.6.2 Producer**

For the producer, to use the SCM approach, we can reuse the same mechanism we use for external contracts. We route Spring Cloud Contract to use the SCM implementation via the URL that contains the git:// protocol.

|  |  |
| --- | --- |
| [Important] | **Important** |
| You have to manually add the pushStubsToScm goal in Maven or execute (bind) the pushStubsToScm task in Gradle. We don’t push stubs to origin of your git repository out of the box. |

**Maven.**

<plugin>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions>

<configuration>

*<!-- Base class mappings etc. -->*

*<!-- We want to pick contracts from a Git repository -->*

<contractsRepositoryUrl>git://https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs-contracts-git.git</contractsRepositoryUrl>

*<!-- We reuse the contract dependency section to set up the path*

*to the folder that contains the contract definitions. In our case the*

*path will be /groupId/artifactId/version/contracts -->*

<contractDependency>

<groupId>${project.groupId}</groupId>

<artifactId>${project.artifactId}</artifactId>

<version>${project.version}</version>

</contractDependency>

*<!-- The contracts mode can't be classpath -->*

<contractsMode>REMOTE</contractsMode>

</configuration>

<executions>

<execution>

<phase>package</phase>

<goals>

*<!-- By default we will not push the stubs back to SCM,*

*you have to explicitly add it as a goal -->*

<goal>pushStubsToScm</goal>

</goals>

</execution>

</executions>

</plugin>

**Gradle.**

contracts {

// We want to pick contracts from a Git repository

contractDependency {

stringNotation = "${project.group}:${project.name}:${project.version}"

}

/\*

We reuse the contract dependency section to set up the path

to the folder that contains the contract definitions. In our case the

path will be /groupId/artifactId/version/contracts

\*/

contractRepository {

repositoryUrl = "git://https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs-contracts-git.git"

}

// The mode can't be classpath

contractsMode = "REMOTE"

// Base class mappings etc.

}

/\*

In this scenario we want to publish stubs to SCM whenever

the `publish` task is executed

\*/

publish.dependsOn("publishStubsToScm")

With such a setup:

* Git project will be cloned to a temporary directory
* The SCM stub downloader will go to META-INF/groupId/artifactId/version/contracts folder to find contracts. E.g. for com.example:foo:1.0.0 the path would be META-INF/com.example/foo/1.0.0/contracts
* Tests will be generated from the contracts
* Stubs will be created from the contracts
* Once the tests pass, the stubs will be committed in the cloned repository
* Finally, a push will be done to that repo’s origin

**90.6.3 Consumer**

On the consumer side when passing the repositoryRoot parameter, either from the @AutoConfigureStubRunner annotation, the JUnit rule or properties, it’s enough to pass the URL of the SCM repository, prefixed with the protocol. For example

@AutoConfigureStubRunner(

stubsMode="REMOTE",

repositoryRoot="git://https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs-contracts-git.git",

ids="com.example:bookstore:0.0.1.RELEASE"

)

With such a setup:

* Git project will be cloned to a temporary directory
* The SCM stub downloader will go to META-INF/groupId/artifactId/version/ folder to find stub definitions and contracts. E.g. for com.example:foo:1.0.0the path would be META-INF/com.example/foo/1.0.0/
* Stub servers will be started and fed with mappings
* Messaging definitions will be read and used in the messaging tests

**90.7 Can I use the Pact Broker?**

When using [Pact](http://pact.io/) you can use the [Pact Broker](https://github.com/pact-foundation/pact_broker) to store and share Pact definitions. Starting from Spring Cloud Contract 2.0.0 one can fetch Pact files from the Pact Broker to generate tests and stubs.

As a prerequisite the Pact Converter and Pact Stub Downloader are required. You have to add it via the spring-cloud-contract-pact dependency. You can read more about it in the [Section 97.1.1, “Pact Converter”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#pact-converter) section.

|  |  |
| --- | --- |
| [Important] | **Important** |
| Pact follows the Consumer Contract convention. That means that the Consumer creates the Pact definitions first, then shares the files with the Producer. Those expectations are generated from the Consumer’s code and can break the Producer if the expectation is not met. |

**90.7.1 Pact Consumer**

The consumer uses Pact framework to generate Pact files. The Pact files are sent to the Pact Broker. An example of such setup can be found [here](https://github.com/spring-cloud-samples/spring-cloud-contract-samples/tree/2.0.x/consumer_pact).

**90.7.2 Producer**

For the producer, to use the Pact files from the Pact Broker, we can reuse the same mechanism we use for external contracts. We route Spring Cloud Contract to use the Pact implementation via the URL that contains the pact:// protocol. It’s enough to pass the URL to the Pact Broker. An example of such setup can be found [here](https://github.com/spring-cloud-samples/spring-cloud-contract-samples/tree/2.0.x/producer_pact).

**Maven.**

<plugin>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions>

<configuration>

*<!-- Base class mappings etc. -->*

*<!-- We want to pick contracts from a Git repository -->*

<contractsRepositoryUrl>pact://http://localhost:8085</contractsRepositoryUrl>

*<!-- We reuse the contract dependency section to set up the path*

*to the folder that contains the contract definitions. In our case the*

*path will be /groupId/artifactId/version/contracts -->*

<contractDependency>

<groupId>${project.groupId}</groupId>

<artifactId>${project.artifactId}</artifactId>

*<!-- When + is passed, a latest tag will be applied when fetching pacts -->*

<version>+</version>

</contractDependency>

*<!-- The contracts mode can't be classpath -->*

<contractsMode>REMOTE</contractsMode>

</configuration>

*<!-- Don't forget to add spring-cloud-contract-pact to the classpath! -->*

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-contract-pact</artifactId>

<version>${spring-cloud-contract.version}</version>

</dependency>

</dependencies>

</plugin>

**Gradle.**

buildscript {

repositories {

//...

}

dependencies {

// ...

// Don't forget to add spring-cloud-contract-pact to the classpath!

classpath "org.springframework.cloud:spring-cloud-contract-pact:${contractVersion}"

}

}

contracts {

// When + is passed, a latest tag will be applied when fetching pacts

contractDependency {

stringNotation = "${project.group}:${project.name}:+"

}

contractRepository {

repositoryUrl = "pact://http://localhost:8085"

}

// The mode can't be classpath

contractsMode = "REMOTE"

// Base class mappings etc.

}

With such a setup:

* Pact files will be downloaded from the Pact Broker
* Spring Cloud Contract will convert the Pact files into tests and stubs
* The JAR with the stubs gets automatically created as usual

**90.7.3 Pact Consumer (Producer Contract approach)**

In the scenario where you don’t want to do Consumer Contract approach (for every single consumer define the expectations) but you’d prefer to do Producer Contracts (the producer provides the contracts and publishes stubs), it’s enough to use Spring Cloud Contract with Stub Runner option. An example of such setup can be found [here](https://github.com/spring-cloud-samples/spring-cloud-contract-samples/tree/2.0.x/consumer_pact_stubrunner).

First, remember to add Stub Runner and Spring Cloud Contract Pact module as test dependencies.

**Maven.**

<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>

*<!-- Don't forget to add spring-cloud-contract-pact to the classpath! -->*

<dependencies>

*<!-- ... -->*

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-contract-stub-runner</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-contract-pact</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

**Gradle.**

dependencyManagement {

imports {

mavenBom "org.springframework.cloud:spring-cloud-dependencies:${springCloudVersion}"

}

}

dependencies {

//...

testCompile("org.springframework.cloud:spring-cloud-starter-contract-stub-runner")

// Don't forget to add spring-cloud-contract-pact to the classpath!

testCompile("org.springframework.cloud:spring-cloud-contract-pact")

}

Next, just pass the URL of the Pact Broker to repositoryRoot, prefixed with pact:// protocol. E.g. pact://http://localhost:8085

*@RunWith(SpringRunner.class)*

*@SpringBootTest*

*@AutoConfigureStubRunner(stubsMode = StubRunnerProperties.StubsMode.REMOTE,*

*ids = "com.example:beer-api-producer-pact",*

*repositoryRoot = "pact://http://localhost:8085")*

**public** **class** BeerControllerTest {

*//Inject the port of the running stub*

*@StubRunnerPort("beer-api-producer-pact")* **int** producerPort;

*//...*

}

With such a setup:

* Pact files will be downloaded from the Pact Broker
* Spring Cloud Contract will convert the Pact files into stub definitions
* The stub servers will be started and fed with stubs

For more information about Pact support you can go to the [Section 97.7, “Using the Pact Stub Downloader”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#pact-stub-downloader) section.

**90.8 How can I debug the request/response being sent by the generated tests client?**

The generated tests all boil down to RestAssured in some form or fashion which relies on [Apache HttpClient](https://hc.apache.org/httpcomponents-client-ga/). HttpClient has a facility called [wire logging](https://hc.apache.org/httpcomponents-client-ga/logging.html#Wire_Logging) which logs the entire request and response to HttpClient. Spring Boot has a logging [common application property](https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html) for doing this sort of thing, just add this to your application properties

logging.level.org.apache.http.wire=DEBUG

**90.8.1 How can I debug the mapping/request/response being sent by WireMock?**

Starting from version 1.2.0 we turn on WireMock logging to info and the WireMock notifier to being verbose. Now you will exactly know what request was received by WireMock server and which matching response definition was picked.

To turn off this feature just bump WireMock logging to ERROR

logging.level.com.github.tomakehurst.wiremock=ERROR

**90.8.2 How can I see what got registered in the HTTP server stub?**

You can use the mappingsOutputFolder property on @AutoConfigureStubRunner or StubRunnerRule to dump all mappings per artifact id. Also the port at which the given stub server was started will be attached.

**90.8.3 Can I reference text from file?**

Yes! With version 1.2.0 we’ve added such a possibility. It’s enough to call file(…​) method in the DSL and provide a path relative to where the contract lays. If you’re using YAML just use the bodyFromFile property.

**91. Spring Cloud Contract Verifier Setup**

You can set up Spring Cloud Contract Verifier in the following ways:

* [As a Gradle project](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gradle-project)
* [As a Maven project](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#maven-project)
* [As a Docker project](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#docker-project)

**91.1 Gradle Project**

To learn how to set up the Gradle project for Spring Cloud Contract Verifier, read the following sections:

* [Section 91.1.1, “Prerequisites”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gradle-prerequisites)
* [Section 91.1.2, “Add Gradle Plugin with Dependencies”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gradle-add-gradle-plugin)
* [Section 91.1.3, “Gradle and Rest Assured 2.0”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gradle-and-rest-assured)
* [Section 91.1.4, “Snapshot Versions for Gradle”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gradle-snapshot-versions)
* [Section 91.1.5, “Add stubs”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gradle-add-stubs)
* [Section 91.1.7, “Default Setup”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gradle-default-setup)
* [Section 91.1.8, “Configure Plugin”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gradle-configure-plugin)
* [Section 91.1.9, “Configuration Options”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gradle-configuration-options)
* [Section 91.1.10, “Single Base Class for All Tests”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gradle-single-base-class)
* [Section 91.1.11, “Different Base Classes for Contracts”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gradle-different-base-classes)
* [Section 91.1.12, “Invoking Generated Tests”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gradle-invoking-generated-tests)
* [Section 91.1.13, “Pushing stubs to SCM”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gradle-pushing-stubs-to-scm)
* [Section 91.1.14, “Spring Cloud Contract Verifier on the Consumer Side”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gradle-consumer)

**91.1.1 Prerequisites**

In order to use Spring Cloud Contract Verifier with WireMock, you muse use either a Gradle or a Maven plugin.

|  |
| --- |
| [Warning] |
| If you want to use Spock in your projects, you must add separately the spock-core and spock-spring modules. Check [Spock docs for more information](https://spockframework.github.io/) |

**91.1.2 Add Gradle Plugin with Dependencies**

To add a Gradle plugin with dependencies, use code similar to this:

buildscript {

repositories {

mavenCentral()

}

dependencies {

classpath "org.springframework.boot:spring-boot-gradle-plugin:${springboot\_version}"

classpath "org.springframework.cloud:spring-cloud-contract-gradle-plugin:${verifier\_version}"

}

}

apply plugin: 'groovy'

apply plugin: 'spring-cloud-contract'

dependencyManagement {

imports {

mavenBom "org.springframework.cloud:spring-cloud-contract-dependencies:${verifier\_version}"

}

}

dependencies {

testCompile 'org.codehaus.groovy:groovy-all:2.4.6'

*// example with adding Spock core and Spock Spring*

testCompile 'org.spockframework:spock-core:1.0-groovy-2.4'

testCompile 'org.spockframework:spock-spring:1.0-groovy-2.4'

testCompile 'org.springframework.cloud:spring-cloud-starter-contract-verifier'

}

**91.1.3 Gradle and Rest Assured 2.0**

By default, Rest Assured 3.x is added to the classpath. However, to use Rest Assured 2.x you can add it to the plugins classpath, as shown here:

buildscript {

repositories {

mavenCentral()

}

dependencies {

classpath "org.springframework.boot:spring-boot-gradle-plugin:${springboot\_version}"

classpath "org.springframework.cloud:spring-cloud-contract-gradle-plugin:${verifier\_version}"

classpath "com.jayway.restassured:rest-assured:2.5.0"

classpath "com.jayway.restassured:spring-mock-mvc:2.5.0"

}

}

depenendencies {

*// all dependencies*

*// you can exclude rest-assured from spring-cloud-contract-verifier*

testCompile "com.jayway.restassured:rest-assured:2.5.0"

testCompile "com.jayway.restassured:spring-mock-mvc:2.5.0"

}

That way, the plugin automatically sees that Rest Assured 2.x is present on the classpath and modifies the imports accordingly.

**91.1.4 Snapshot Versions for Gradle**

Add the additional snapshot repository to your build.gradle to use snapshot versions, which are automatically uploaded after every successful build, as shown here:

buildscript {

repositories {

mavenCentral()

mavenLocal()

maven { url "http://repo.spring.io/snapshot" }

maven { url "http://repo.spring.io/milestone" }

maven { url "http://repo.spring.io/release" }

}

}

**91.1.5 Add stubs**

By default, Spring Cloud Contract Verifier is looking for stubs in the src/test/resources/contracts directory.

The directory containing stub definitions is treated as a class name, and each stub definition is treated as a single test. Spring Cloud Contract Verifier assumes that it contains at least one level of directories that are to be used as the test class name. If more than one level of nested directories is present, all except the last one is used as the package name. For example, with following structure:

src/test/resources/contracts/myservice/shouldCreateUser.groovy

src/test/resources/contracts/myservice/shouldReturnUser.groovy

Spring Cloud Contract Verifier creates a test class named defaultBasePackage.MyService with two methods:

* shouldCreateUser()
* shouldReturnUser()

**91.1.6 Run the Plugin**

The plugin registers itself to be invoked before a check task. If you want it to be part of your build process, you need to do nothing more. If you just want to generate tests, invoke the generateContractTests task.

**91.1.7 Default Setup**

The default Gradle Plugin setup creates the following Gradle part of the build (in pseudocode):

contracts {

targetFramework = 'JUNIT'

testMode = 'MockMvc'

generatedTestSourcesDir = project.file("${project.buildDir}/generated-test-sources/contracts")

contractsDslDir = "${project.rootDir}/src/test/resources/contracts"

basePackageForTests = 'org.springframework.cloud.verifier.tests'

stubsOutputDir = project.file("${project.buildDir}/stubs")

*// the following properties are used when you want to provide where the JAR with contract lays*

contractDependency {

stringNotation = ''

}

contractsPath = ''

contractsWorkOffline = false

contractRepository {

cacheDownloadedContracts(true)

}

}

tasks.create(type: Jar, name: 'verifierStubsJar', dependsOn: 'generateClientStubs') {

baseName = project.name

classifier = contracts.stubsSuffix

from contractVerifier.stubsOutputDir

}

project.artifacts {

archives task

}

tasks.create(type: Copy, name: 'copyContracts') {

from contracts.contractsDslDir

into contracts.stubsOutputDir

}

verifierStubsJar.dependsOn 'copyContracts'

publishing {

publications {

stubs(MavenPublication) {

artifactId project.name

artifact verifierStubsJar

}

}

}

**91.1.8 Configure Plugin**

To change the default configuration, add a contracts snippet to your Gradle config, as shown here:

contracts {

testMode = 'MockMvc'

baseClassForTests = 'org.mycompany.tests'

generatedTestSourcesDir = project.file('src/generatedContract')

}

**91.1.9 Configuration Options**

* **testMode**: Defines the mode for acceptance tests. By default, the mode is MockMvc, which is based on Spring’s MockMvc. It can also be changed to **JaxRsClient** or to **Explicit** for real HTTP calls.
* **imports**: Creates an array with imports that should be included in generated tests (for example ['org.myorg.Matchers']). By default, it creates an empty array.
* **staticImports**: Creates an array with static imports that should be included in generated tests(for example ['org.myorg.Matchers.\*']). By default, it creates an empty array.
* **basePackageForTests**: Specifies the base package for all generated tests. If not set, the value is picked from baseClassForTests’s package and from `packageWithBaseClasses. If neither of these values are set, then the value is set toorg.springframework.cloud.contract.verifier.tests.
* **baseClassForTests**: Creates a base class for all generated tests. By default, if you use Spock classes, the class is spock.lang.Specification.
* **packageWithBaseClasses**: Defines a package where all the base classes reside. This setting takes precedence over **baseClassForTests**.
* **baseClassMappings**: Explicitly maps a contract package to a FQN of a base class. This setting takes precedence over **packageWithBaseClasses** and **baseClassForTests**.
* **ruleClassForTests**: Specifies a rule that should be added to the generated test classes.
* **ignoredFiles**: Uses an Antmatcher to allow defining stub files for which processing should be skipped. By default, it is an empty array.
* **contractsDslDir**: Specifies the directory containing contracts written using the GroovyDSL. By default, its value is $rootDir/src/test/resources/contracts.
* **generatedTestSourcesDir**: Specifies the test source directory where tests generated from the Groovy DSL should be placed. By default its value is$buildDir/generated-test-sources/contractVerifier.
* **stubsOutputDir**: Specifies the directory where the generated WireMock stubs from the Groovy DSL should be placed.
* **targetFramework**: Specifies the target test framework to be used. Currently, Spock and JUnit are supported with JUnit being the default framework.
* **contractsProperties**: a map containing properties to be passed to Spring Cloud Contract components. Those properties might be used by e.g. inbuilt or custom Stub Downloaders.

The following properties are used when you want to specify the location of the JAR containing the contracts: \* **contractDependency**: Specifies the Dependency that provides groupid:artifactid:version:classifier coordinates. You can use the contractDependency closure to set it up. \* **contractsPath**: Specifies the path to the jar. If contract dependencies are downloaded, the path defaults to groupid/artifactid where groupid is slash separated. Otherwise, it scans contracts under the provided directory. \* **contractsMode**: Specifies the mode of downloading contracts (whether the JAR is available offline, remotely etc.) \* **contractsSnapshotCheckSkip**: If set to true will not assert whether the downloaded stubs / contract JAR was downloaded from a remote location or a local one(only applicable to Maven repos, not Git or Pact). \* **deleteStubsAfterTest**: If set to false will not remove any downloaded contracts from temporary directories

**91.1.10 Single Base Class for All Tests**

When using Spring Cloud Contract Verifier in default MockMvc, you need to create a base specification for all generated acceptance tests. In this class, you need to point to an endpoint, which should be verified.

**abstract** **class** BaseMockMvcSpec **extends** Specification {

def setup() {

RestAssuredMockMvc.standaloneSetup(**new** PairIdController())

}

**void** isProperCorrelationId(Integer correlationId) {

assert correlationId == 123456

}

**void** isEmpty(String value) {

assert value == null

}

}

If you use Explicit mode, you can use a base class to initialize the whole tested app as you might see in regular integration tests. If you use the JAXRSCLIENT mode, this base class should also contain a protected WebTarget webTarget field. Right now, the only option to test the JAX-RS API is to start a web server.

**91.1.11 Different Base Classes for Contracts**

If your base classes differ between contracts, you can tell the Spring Cloud Contract plugin which class should get extended by the autogenerated tests. You have two options:

* Follow a convention by providing the packageWithBaseClasses
* Provide explicit mapping via baseClassMappings

**By Convention**

The convention is such that if you have a contract under (for example) src/test/resources/contract/foo/bar/baz/ and set the value of thepackageWithBaseClasses property to com.example.base, then Spring Cloud Contract Verifier assumes that there is a BarBazBase class under the com.example.base package. In other words, the system takes the last two parts of the package, if they exist, and forms a class with a Base suffix. This rule takes precedence over **baseClassForTests**. Here is an example of how it works in the contracts closure:

packageWithBaseClasses = 'com.example.base'

**By Mapping**

You can manually map a regular expression of the contract’s package to fully qualified name of the base class for the matched contract. You have to provide a list calledbaseClassMappings that consists baseClassMapping objects that takes a contractPackageRegex to baseClassFQN mapping. Consider the following example:

baseClassForTests = "com.example.FooBase"

baseClassMappings {

baseClassMapping('.\*/com/.\*', 'com.example.ComBase')

baseClassMapping('.\*/bar/.\*':'com.example.BarBase')

}

Let’s assume that you have contracts under - src/test/resources/contract/com/ - src/test/resources/contract/foo/

By providing the baseClassForTests, we have a fallback in case mapping did not succeed. (You could also provide the packageWithBaseClasses as a fallback.) That way, the tests generated from src/test/resources/contract/com/ contracts extend the com.example.ComBase, whereas the rest of the tests extend com.example.FooBase.

**91.1.12 Invoking Generated Tests**

To ensure that the provider side is compliant with defined contracts, you need to invoke:

./gradlew generateContractTests **test**

**91.1.13 Pushing stubs to SCM**

If you’re using the SCM repository to keep the contracts and stubs, you might want to automate the step of pushing stubs to the repository. To do that, it’s enough to call the pushStubsToScm task. Example:

$ ./gradlew pushStubsToScm

Under [Section 97.6, “Using the SCM Stub Downloader”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#scm-stub-downloader) you can find all possible configuration options that you can pass either via the contractsProperties field e.g. contracts { contractsProperties = [foo:"bar"] }, via contractsProperties method e.g. contracts { contractsProperties([foo:"bar"]) }, a system property or an environment variable.

**91.1.14 Spring Cloud Contract Verifier on the Consumer Side**

In a consuming service, you need to configure the Spring Cloud Contract Verifier plugin in exactly the same way as in case of provider. If you do not want to use Stub Runner then you need to copy contracts stored in src/test/resources/contracts and generate WireMock JSON stubs using:

./gradlew generateClientStubs

|  |
| --- |
| [Note] |
| The stubsOutputDir option has to be set for stub generation to work. |

When present, JSON stubs can be used in automated tests of consuming a service.

*@ContextConfiguration(loader == SpringApplicationContextLoader, classes == Application)*

**class** LoanApplicationServiceSpec **extends** Specification {

*@ClassRule*

*@Shared*

WireMockClassRule wireMockRule == **new** WireMockClassRule()

*@Autowired*

LoanApplicationService sut

def 'should successfully apply for loan'() {

given:

LoanApplication application =

**new** LoanApplication(client: **new** Client(clientPesel: '12345678901'), amount: 123.123)

when:

LoanApplicationResult loanApplication == sut.loanApplication(application)

then:

loanApplication.loanApplicationStatus == LoanApplicationStatus.LOAN\_APPLIED

loanApplication.rejectionReason == null

}

}

LoanApplication makes a call to FraudDetection service. This request is handled by a WireMock server configured with stubs generated by Spring Cloud Contract Verifier.

**91.2 Maven Project**

To learn how to set up the Maven project for Spring Cloud Contract Verifier, read the following sections:

* [Section 91.2.1, “Add maven plugin”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#maven-add-plugin)
* [Section 91.2.2, “Maven and Rest Assured 2.0”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#maven-rest-assured)
* [Section 91.2.3, “Snapshot versions for Maven”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#maven-snapshot-versions)
* [Section 91.2.4, “Add stubs”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#maven-add-stubs)
* [Section 91.2.5, “Run plugin”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#maven-run-plugin)
* [Section 91.2.6, “Configure plugin”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#maven-configure-plugin)
* [Section 91.2.7, “Configuration Options”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#maven-configuration-options)
* [Section 91.2.8, “Single Base Class for All Tests”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#maven-single-base)
* [Section 91.2.9, “Different base classes for contracts”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#maven-different-base)
* [Section 91.2.10, “Invoking generated tests”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#maven-invoking-generated-tests)
* [Section 91.2.11, “Pushing stubs to SCM”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#maven-pushing-stubs-to-scm)
* [Section 91.2.12, “Maven Plugin and STS”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#maven-sts)

**91.2.1 Add maven plugin**

Add the Spring Cloud Contract BOM in a fashion similar to this:

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>${spring-cloud-dependencies.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

Next, add the Spring Cloud Contract Verifier Maven plugin:

<plugin>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions>

<configuration>

<packageWithBaseClasses>com.example.fraud</packageWithBaseClasses>

</configuration>

</plugin>

You can read more in the [Spring Cloud Contract Maven Plugin Documentation (example for 2.0.0.RELEASE version)](https://cloud.spring.io/spring-cloud-static/spring-cloud-contract/2.0.0.RELEASE/spring-cloud-contract-maven-plugin/).

**91.2.2 Maven and Rest Assured 2.0**

By default, Rest Assured 3.x is added to the classpath. However, you can use Rest Assured 2.x by adding it to the plugins classpath, as shown here:

<plugin>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions>

<configuration>

<packageWithBaseClasses>com.example</packageWithBaseClasses>

</configuration>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-contract-verifier</artifactId>

<version>${spring-cloud-contract.version}</version>

</dependency>

<dependency>

<groupId>com.jayway.restassured</groupId>

<artifactId>rest-assured</artifactId>

<version>2.5.0</version>

<scope>compile</scope>

</dependency>

<dependency>

<groupId>com.jayway.restassured</groupId>

<artifactId>spring-mock-mvc</artifactId>

<version>2.5.0</version>

<scope>compile</scope>

</dependency>

</dependencies>

</plugin>

<dependencies>

<!-- all dependencies -->

<!-- you can exclude rest-assured from spring-cloud-contract-verifier -->

<dependency>

<groupId>com.jayway.restassured</groupId>

<artifactId>rest-assured</artifactId>

<version>2.5.0</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>com.jayway.restassured</groupId>

<artifactId>spring-mock-mvc</artifactId>

<version>2.5.0</version>

<scope>test</scope>

</dependency>

</dependencies>

That way, the plugin automatically sees that Rest Assured 3.x is present on the classpath and modifies the imports accordingly.

**91.2.3 Snapshot versions for Maven**

For Snapshot and Milestone versions, you have to add the following section to your pom.xml, as shown here:

<repositories>

<repository>

<id>spring-snapshots</id>

<name>Spring Snapshots</name>

<url>https://repo.spring.io/snapshot</url>

<snapshots>

<enabled>true</enabled>

</snapshots>

</repository>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

<repository>

<id>spring-releases</id>

<name>Spring Releases</name>

<url>https://repo.spring.io/release</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

<pluginRepositories>

<pluginRepository>

<id>spring-snapshots</id>

<name>Spring Snapshots</name>

<url>https://repo.spring.io/snapshot</url>

<snapshots>

<enabled>true</enabled>

</snapshots>

</pluginRepository>

<pluginRepository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</pluginRepository>

<pluginRepository>

<id>spring-releases</id>

<name>Spring Releases</name>

<url>https://repo.spring.io/release</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</pluginRepository>

</pluginRepositories>

**91.2.4 Add stubs**

By default, Spring Cloud Contract Verifier is looking for stubs in the src/test/resources/contracts directory. The directory containing stub definitions is treated as a class name, and each stub definition is treated as a single test. We assume that it contains at least one directory to be used as test class name. If there is more than one level of nested directories, all except the last one is used as package name. For example, with following structure:

src/test/resources/contracts/myservice/shouldCreateUser.groovy

src/test/resources/contracts/myservice/shouldReturnUser.groovy

Spring Cloud Contract Verifier creates a test class named defaultBasePackage.MyService with two methods

* shouldCreateUser()
* shouldReturnUser()

**91.2.5 Run plugin**

The plugin goal generateTests is assigned to be invoked in the phase called generate-test-sources. If you want it to be part of your build process, you need not do anything. If you just want to generate tests, invoke the generateTests goal.

**91.2.6 Configure plugin**

To change the default configuration, just add a configuration section to the plugin definition or the execution definition, as shown here:

<plugin>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<executions>

<execution>

<goals>

<goal>convert</goal>

<goal>generateStubs</goal>

<goal>generateTests</goal>

</goals>

</execution>

</executions>

<configuration>

<basePackageForTests>org.springframework.cloud.verifier.twitter.place</basePackageForTests>

<baseClassForTests>org.springframework.cloud.verifier.twitter.place.BaseMockMvcSpec</baseClassForTests>

</configuration>

</plugin>

**91.2.7 Configuration Options**

* **testMode**: Defines the mode for acceptance tests. By default, the mode is MockMvc, which is based on Spring’s MockMvc. It can also be changed to **JaxRsClient** or to **Explicit** for real HTTP calls.
* **basePackageForTests**: Specifies the base package for all generated tests. If not set, the value is picked from baseClassForTests’s package and from `packageWithBaseClasses. If neither of these values are set, then the value is set toorg.springframework.cloud.contract.verifier.tests.
* **ruleClassForTests**: Specifies a rule that should be added to the generated test classes.
* **baseClassForTests**: Creates a base class for all generated tests. By default, if you use Spock classes, the class is spock.lang.Specification.
* **contractsDirectory**: Specifies a directory containing contracts written with the GroovyDSL. The default directory is /src/test/resources/contracts.
* **testFramework**: Specifies the target test framework to be used. Currently, Spock and JUnit are supported with JUnit being the default framework
* **packageWithBaseClasses**: Defines a package where all the base classes reside. This setting takes precedence over **baseClassForTests**. The convention is such that, if you have a contract under (for example) src/test/resources/contract/foo/bar/baz/ and set the value of the packageWithBaseClasses property to com.example.base, then Spring Cloud Contract Verifier assumes that there is a BarBazBase class under the com.example.base package. In other words, the system takes the last two parts of the package, if they exist, and forms a class with a Base suffix.
* **baseClassMappings**: Specifies a list of base class mappings that provide contractPackageRegex, which is checked against the package where the contract is located, and baseClassFQN, which maps to the fully qualified name of the base class for the matched contract. For example, if you have a contract undersrc/test/resources/contract/foo/bar/baz/ and map the property .\* → com.example.base.BaseClass, then the test class generated from these contracts extends com.example.base.BaseClass. This setting takes precedence over **packageWithBaseClasses** and **baseClassForTests**.
* **contractsProperties**: a map containing properties to be passed to Spring Cloud Contract components. Those properties might be used by e.g. inbuilt or custom Stub Downloaders.

If you want to download your contract definitions from a Maven repository, you can use the following options:

* **contractDependency**: The contract dependency that contains all the packaged contracts.
* **contractsPath**: The path to the concrete contracts in the JAR with packaged contracts. Defaults to groupid/artifactid where gropuid is slash separated.
* **contractsMode**: Picks the mode in which stubs will be found and registered
* **contractsSnapshotCheckSkip**: If true then will not assert whether a stub / contract JAR was downloaded from local or remote location
* **deleteStubsAfterTest**: If set to false will not remove any downloaded contracts from temporary directories
* **contractsRepositoryUrl**: URL to a repo with the artifacts that have contracts. If it is not provided, use the current Maven ones.
* **contractsRepositoryUsername**: The user name to be used to connect to the repo with contracts.
* **contractsRepositoryPassword**: The password to be used to connect to the repo with contracts.
* **contractsRepositoryProxyHost**: The proxy host to be used to connect to the repo with contracts.
* **contractsRepositoryProxyPort**: The proxy port to be used to connect to the repo with contracts.

We cache only non-snapshot, explicitly provided versions (for example + or 1.0.0.BUILD-SNAPSHOT won’t get cached). By default, this feature is turned on.

**91.2.8 Single Base Class for All Tests**

When using Spring Cloud Contract Verifier in default MockMvc, you need to create a base specification for all generated acceptance tests. In this class, you need to point to an endpoint, which should be verified.

**package** org.mycompany.tests

**import** org.mycompany.ExampleSpringController

**import** com.jayway.restassured.module.mockmvc.RestAssuredMockMvc

**import** spock.lang.Specification

**class** MvcSpec **extends** Specification {

def setup() {

RestAssuredMockMvc.standaloneSetup(**new** ExampleSpringController())

}

}

You can also setup the whole context if necessary.

**import** io.restassured.module.mockmvc.RestAssuredMockMvc;

**import** org.junit.Before;

**import** org.junit.runner.RunWith;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.boot.test.context.SpringBootTest;

**import** org.springframework.test.context.junit4.SpringRunner;

**import** org.springframework.web.context.WebApplicationContext;

*@RunWith(SpringRunner.class)*

*@SpringBootTest(webEnvironment = WebEnvironment.RANDOM\_PORT, classes = SomeConfig.class, properties="some=property")*

**public** **abstract** **class** BaseTestClass {

*@Autowired*

WebApplicationContext context;

*@Before*

**public** **void** setup() {

RestAssuredMockMvc.webAppContextSetup(**this**.context);

}

}

If you use EXPLICIT mode, you can use a base class to initialize the whole tested app similarly, as you might find in regular integration tests.

**import** io.restassured.RestAssured;

**import** org.junit.Before;

**import** org.junit.runner.RunWith;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.boot.test.context.SpringBootTest;

**import** org.springframework.boot.web.server.LocalServerPort

**import** org.springframework.test.context.junit4.SpringRunner;

**import** org.springframework.web.context.WebApplicationContext;

*@RunWith(SpringRunner.class)*

*@SpringBootTest(webEnvironment = WebEnvironment.RANDOM\_PORT, classes = SomeConfig.class, properties="some=property")*

**public** **abstract** **class** BaseTestClass {

*@LocalServerPort*

**int** port;

*@Before*

**public** **void** setup() {

RestAssured.baseURI = "http://localhost:" + **this**.port;

}

}

If you use the JAXRSCLIENT mode, this base class should also contain a protected WebTarget webTarget field. Right now, the only option to test the JAX-RS API is to start a web server.

**91.2.9 Different base classes for contracts**

If your base classes differ between contracts, you can tell the Spring Cloud Contract plugin which class should get extended by the autogenerated tests. You have two options:

* Follow a convention by providing the packageWithBaseClasses
* provide explicit mapping via baseClassMappings

**By Convention**

The convention is such that if you have a contract under (for example) src/test/resources/contract/foo/bar/baz/ and set the value of thepackageWithBaseClasses property to com.example.base, then Spring Cloud Contract Verifier assumes that there is a BarBazBase class under the com.example.base package. In other words, the system takes the last two parts of the package, if they exist, and forms a class with a Base suffix. This rule takes precedence over **baseClassForTests**. Here is an example of how it works in the contracts closure:

<plugin>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<configuration>

<packageWithBaseClasses>hello</packageWithBaseClasses>

</configuration>

</plugin>

**By Mapping**

You can manually map a regular expression of the contract’s package to fully qualified name of the base class for the matched contract. You have to provide a list calledbaseClassMappings that consists baseClassMapping objects that takes a contractPackageRegex to baseClassFQN mapping. Consider the following example:

<plugin>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<configuration>

<baseClassForTests>com.example.FooBase</baseClassForTests>

<baseClassMappings>

<baseClassMapping>

<contractPackageRegex>.\*com.\*</contractPackageRegex>

<baseClassFQN>com.example.TestBase</baseClassFQN>

</baseClassMapping>

</baseClassMappings>

</configuration>

</plugin>

Assume that you have contracts under these two locations: \* src/test/resources/contract/com/ \* src/test/resources/contract/foo/

By providing the baseClassForTests, we have a fallback in case mapping did not succeed. (You can also provide the packageWithBaseClasses as a fallback.) That way, the tests generated from src/test/resources/contract/com/ contracts extend the com.example.ComBase, whereas the rest of the tests extend com.example.FooBase.

**91.2.10 Invoking generated tests**

The Spring Cloud Contract Maven Plugin generates verification code in a directory called /generated-test-sources/contractVerifier and attaches this directory to testCompile goal.

For Groovy Spock code, use the following:

<plugin>

<groupId>org.codehaus.gmavenplus</groupId>

<artifactId>gmavenplus-plugin</artifactId>

<version>1.5</version>

<executions>

<execution>

<goals>

<goal>testCompile</goal>

</goals>

</execution>

</executions>

<configuration>

<testSources>

<testSource>

<directory>${project.basedir}/src/test/groovy</directory>

<includes>

<include>\*\*/\*.groovy</include>

</includes>

</testSource>

<testSource>

<directory>${project.build.directory}/generated-test-sources/contractVerifier</directory>

<includes>

<include>\*\*/\*.groovy</include>

</includes>

</testSource>

</testSources>

</configuration>

</plugin>

To ensure that provider side is compliant with defined contracts, you need to invoke mvn generateTest test.

**91.2.11 Pushing stubs to SCM**

If you’re using the SCM repository to keep the contracts and stubs, you might want to automate the step of pushing stubs to the repository. To do that, it’s enough to add the pushStubsToScm goal. Example:

<plugin>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions>

<configuration>

*<!-- Base class mappings etc. -->*

*<!-- We want to pick contracts from a Git repository -->*

<contractsRepositoryUrl>git://https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs-contracts-git.git</contractsRepositoryUrl>

*<!-- We reuse the contract dependency section to set up the path*

*to the folder that contains the contract definitions. In our case the*

*path will be /groupId/artifactId/version/contracts -->*

<contractDependency>

<groupId>${project.groupId}</groupId>

<artifactId>${project.artifactId}</artifactId>

<version>${project.version}</version>

</contractDependency>

*<!-- The contracts mode can't be classpath -->*

<contractsMode>REMOTE</contractsMode>

</configuration>

<executions>

<execution>

<phase>package</phase>

<goals>

*<!-- By default we will not push the stubs back to SCM,*

*you have to explicitly add it as a goal -->*

<goal>pushStubsToScm</goal>

</goals>

</execution>

</executions>

</plugin>

Under [Section 97.6, “Using the SCM Stub Downloader”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#scm-stub-downloader) you can find all possible configuration options that you can pass either via the <configuration><contractProperties> map, a system property or an environment variable.

**91.2.12 Maven Plugin and STS**

If you see the following exception while using STS:

When you click on the error marker you should see something like this:

plugin:1.1.0.M1:convert:default-convert:process-**test**-resources) org.apache.maven.plugin.PluginExecutionException: Execution default-convert of goal org.springframework.cloud:spring-

cloud-contract-maven-plugin:1.1.0.M1:convert failed. at org.apache.maven.plugin.DefaultBuildPluginManager.executeMojo(DefaultBuildPluginManager.java:145) at

org.eclipse.m2e.core.internal.embedder.MavenImpl.execute(MavenImpl.java:331) at org.eclipse.m2e.core.internal.embedder.MavenImpl$11.call(MavenImpl.java:1362) at

...

org.eclipse.core.internal.jobs.Worker.run(Worker.java:55) Caused by: java.lang.NullPointerException at

org.eclipse.m2e.core.internal.builder.plexusbuildapi.EclipseIncrementalBuildContext.hasDelta(EclipseIncrementalBuildContext.java:53) at

org.sonatype.plexus.build.incremental.ThreadBuildContext.hasDelta(ThreadBuildContext.java:59) at

In order to fix this issue, provide the following section in your pom.xml:

<build>

<pluginManagement>

<plugins>

*<!--This plugin's configuration is used to store Eclipse m2e settings*

*only. It has no influence on the Maven build itself. -->*

<plugin>

<groupId>org.eclipse.m2e</groupId>

<artifactId>lifecycle-mapping</artifactId>

<version>1.0.0</version>

<configuration>

<lifecycleMappingMetadata>

<pluginExecutions>

<pluginExecution>

<pluginExecutionFilter>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<versionRange>[1.0,)</versionRange>

<goals>

<goal>convert</goal>

</goals>

</pluginExecutionFilter>

<action>

<execute />

</action>

</pluginExecution>

</pluginExecutions>

</lifecycleMappingMetadata>

</configuration>

</plugin>

</plugins>

</pluginManagement>

</build>

**91.3 Stubs and Transitive Dependencies**

The Maven and Gradle plugin that add the tasks that create the stubs jar for you. One problem that arises is that, when reusing the stubs, you can mistakenly import all of that stub’s dependencies. When building a Maven artifact, even though you have a couple of different jars, all of them share one pom:

├── github-webhook-0.0.1.BUILD-20160903.075506-1-stubs.jar

├── github-webhook-0.0.1.BUILD-20160903.075506-1-stubs.jar.sha1

├── github-webhook-0.0.1.BUILD-20160903.075655-2-stubs.jar

├── github-webhook-0.0.1.BUILD-20160903.075655-2-stubs.jar.sha1

├── github-webhook-0.0.1.BUILD-SNAPSHOT.jar

├── github-webhook-0.0.1.BUILD-SNAPSHOT.pom

├── github-webhook-0.0.1.BUILD-SNAPSHOT-stubs.jar

├── ...

└── ...

There are three possibilities of working with those dependencies so as not to have any issues with transitive dependencies:

* Mark all application dependencies as optional
* Create a separate artifactid for the stubs
* Exclude dependencies on the consumer side

**Mark all application dependencies as optional**

If, in the github-webhook application, you mark all of your dependencies as optional, when you include the github-webhook stubs in another application (or when that dependency gets downloaded by Stub Runner) then, since all of the dependencies are optional, they will not get downloaded.

**Create a separate artifactid for the stubs**

If you create a separate artifactid, then you can set it up in whatever way you wish. For example, you might decide to have no dependencies at all.

**Exclude dependencies on the consumer side**

As a consumer, if you add the stub dependency to your classpath, you can explicitly exclude the unwanted dependencies.

**91.4 CI Server setup**

When fetching stubs / contracts in a CI, shared environment, what might happen is that both the producer and the consumer reuse the same local Maven repository. Due to this, the framework, responsible for downloading a stub JAR from remote location, can’t decide which JAR should be picked, local or remote one. That caused the "The artifact was found in the local repository but you have explicitly stated that it should be downloaded from a remote one" exception and failed the build.

For such cases we’re introducing the property and plugin setup mechanism:

* via stubrunner.snapshot-check-skip system property
* via STUBRUNNER\_SNAPSHOT\_CHECK\_SKIP environment variable

if either of these values is set to true, then the stub downloader will not verify the origin of the downloaded JAR.

For the plugins you need to set the contractsSnapshotCheckSkip property to true.

**91.5 Scenarios**

You can handle scenarios with Spring Cloud Contract Verifier. All you need to do is to stick to the proper naming convention while creating your contracts. The convention requires including an order number followed by an underscore. This will work regardles of whether you’re working with YAML or Groovy. Example:

my\_contracts\_dir\

scenario1\

1\_login.groovy

2\_showCart.groovy

3\_logout.groovy

Such a tree causes Spring Cloud Contract Verifier to generate WireMock’s scenario with a name of scenario1 and the three following steps:

1. login marked as Started pointing to…​
2. showCart marked as Step1 pointing to…​
3. logout marked as Step2 which will close the scenario.

More details about WireMock scenarios can be found at <http://wiremock.org/stateful-behaviour.html>

Spring Cloud Contract Verifier also generates tests with a guaranteed order of execution.

**91.6 Docker Project**

We’re publishing a springcloud/spring-cloud-contract Docker image that contains a project that will generate tests and execute them in EXPLICIT mode against a running application.

|  |
| --- |
| [Tip] |
| The EXPLICIT mode means that the tests generated from contracts will send real requests and not the mocked ones. |

**91.6.1 Short intro to Maven, JARs and Binary storage**

Since the Docker image can be used by non JVM projects, it’s good to explain the basic terms behind Spring Cloud Contract packaging defaults.

Part of the following definitions were taken from the [Maven Glossary](https://maven.apache.org/glossary.html)

* Project: Maven thinks in terms of projects. Everything that you will build are projects. Those projects follow a well defined “Project Object Model”. Projects can depend on other projects, in which case the latter are called “dependencies”. A project may consistent of several subprojects, however these subprojects are still treated equally as projects.
* Artifact: An artifact is something that is either produced or used by a project. Examples of artifacts produced by Maven for a project include: JARs, source and binary distributions. Each artifact is uniquely identified by a group id and an artifact ID which is unique within a group.
* JAR: JAR stands for Java ARchive. It’s a format based on the ZIP file format. Spring Cloud Contract packages the contracts and generated stubs in a JAR file.
* GroupId: A group ID is a universally unique identifier for a project. While this is often just the project name (eg. commons-collections), it is helpful to use a fully-qualified package name to distinguish it from other projects with a similar name (eg. org.apache.maven). Typically, when published to the Artifact Manager, the GroupId will get slash separated and form part of the URL. E.g. for group id com.example and artifact id application would be /com/example/application/.
* Classifier: The Maven dependency notation looks as follows: groupId:artifactId:version:classifier. The classifier is additional suffix passed to the dependency. E.g. stubs, sources. The same dependency e.g. com.example:application can produce multiple artifacts that differ from each other with the classifier.
* Artifact manager: When you generate binaries / sources / packages, you would like them to be available for others to download / reference or reuse. In case of the JVM world those artifacts would be JARs, for Ruby these are gems and for Docker those would be Docker images. You can store those artifacts in a manager. Examples of such managers can be [Artifactory](https://jfrog.com/artifactory/) or [Nexus](http://www.sonatype.org/nexus/).

**91.6.2 How it works**

The image searches for contracts under the /contracts folder. The output from running the tests will be available under /spring-cloud-contract/build folder (it’s useful for debugging purposes).

It’s enough for you to mount your contracts, pass the environment variables and the image will:

* generate the contract tests
* execute the tests against the provided URL
* generate the [WireMock](http://wiremock.org/) stubs
* (optional - turned on by default) publish the stubs to a Artifact Manager

**Environment Variables**

The Docker image requires some environment variables to point to your running application, to the Artifact manager instance etc.

* PROJECT\_GROUP - your project’s group id. Defaults to com.example
* PROJECT\_VERSION - your project’s version. Defaults to 0.0.1-SNAPSHOT
* PROJECT\_NAME - artifact id. Defaults to example
* REPO\_WITH\_BINARIES\_URL - URL of your Artifact Manager. Defaults to <http://localhost:8081/artifactory/libs-release-local> which is the default URL of [Artifactory](https://jfrog.com/artifactory/) running locally
* REPO\_WITH\_BINARIES\_USERNAME - (optional) username when the Artifact Manager is secured
* REPO\_WITH\_BINARIES\_PASSWORD - (optional) password when the Artifact Manager is secured
* PUBLISH\_ARTIFACTS - if set to true then will publish artifact to binary storage. Defaults to true.

These environment variables are used when contracts lay in an external repository. To enable this feature you must set the EXTERNAL\_CONTRACTS\_ARTIFACT\_IDenvironment variable.

* EXTERNAL\_CONTRACTS\_GROUP\_ID - group id of the project with contracts. Defaults to com.example
* EXTERNAL\_CONTRACTS\_ARTIFACT\_ID- artifact id of the project with contracts.
* EXTERNAL\_CONTRACTS\_CLASSIFIER- classifier of the project with contracts. Empty by default
* EXTERNAL\_CONTRACTS\_VERSION - version of the project with contracts. Defaults to +, equivalent to picking the latest
* EXTERNAL\_CONTRACTS\_REPO\_WITH\_BINARIES\_URL - URL of your Artifact Manager. Defaults to value of REPO\_WITH\_BINARIES\_URL env var. If that’s not set, defaults to <http://localhost:8081/artifactory/libs-release-local> which is the default URL of [Artifactory](https://jfrog.com/artifactory/) running locally
* EXTERNAL\_CONTRACTS\_PATH - path to contracts for the given project, inside the project with contracts. Defaults to slash separated EXTERNAL\_CONTRACTS\_GROUP\_ID concatenated with / and EXTERNAL\_CONTRACTS\_ARTIFACT\_ID. E.g. for group id foo.bar and artifact id baz, would result in foo/bar/baz contracts path.
* EXTERNAL\_CONTRACTS\_WORK\_OFFLINE - if set to true then will retrieve artifact with contracts from the container’s .m2. Mount your local .m2 as a volume available at the container’s /root/.m2 path. You must not set both EXTERNAL\_CONTRACTS\_WORK\_OFFLINE and EXTERNAL\_CONTRACTS\_REPO\_WITH\_BINARIES\_URL.

These environment variables are used when tests are executed:

* APPLICATION\_BASE\_URL - url against which tests should be executed. Remember that it has to be accessible from the Docker container (e.g. localhost will not work)
* APPLICATION\_USERNAME - (optional) username for basic authentication to your application
* APPLICATION\_PASSWORD - (optional) password for basic authentication to your application

**91.6.3 Example of usage**

Let’s take a look at a simple MVC application

$ git clone https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs

$ **cd** bookstore

The contracts are available under /contracts folder.

**91.6.4 Server side (nodejs)**

Since we want to run tests, we could just execute:

$ npm **test**

however, for learning purposes, let’s split it into pieces:

*# Stop docker infra (nodejs, artifactory)*

$ ./stop\_infra.sh

*# Start docker infra (nodejs, artifactory)*

$ ./setup\_infra.sh

*# Kill & Run app*

$ pkill -f "node app"

$ nohup node app &

*# Prepare environment variables*

$ SC\_CONTRACT\_DOCKER\_VERSION="..."

$ APP\_IP="192.168.0.100"

$ APP\_PORT="3000"

$ ARTIFACTORY\_PORT="8081"

$ APPLICATION\_BASE\_URL="http://${APP\_IP}:${APP\_PORT}"

$ ARTIFACTORY\_URL="http://${APP\_IP}:${ARTIFACTORY\_PORT}/artifactory/libs-release-local"

$ CURRENT\_DIR="$( pwd )"

$ CURRENT\_FOLDER\_NAME=${PWD*##\*/}*

$ PROJECT\_VERSION="0.0.1.RELEASE"

*# Execute contract tests*

$ docker run --rm -e "APPLICATION\_BASE\_URL=${APPLICATION\_BASE\_URL}" -e "PUBLISH\_ARTIFACTS=true" -e "PROJECT\_NAME=${CURRENT\_FOLDER\_NAME}" -e "REPO\_WITH\_BINARIES\_URL=${ARTIFACTORY\_URL}" -e "PROJECT\_VERSION=${PROJECT\_VERSION}" -v "${CURRENT\_DIR}/contracts/:/contracts:ro" -v "${CURRENT\_DIR}/node\_modules/spring-cloud-contract/output:/spring-cloud-contract-output/" springcloud/spring-cloud-contract:"${SC\_CONTRACT\_DOCKER\_VERSION}"

*# Kill app*

$ pkill -f "node app"

What will happen is that via bash scripts:

* infrastructure will be set up (MongoDb, Artifactory). In real life scenario you would just run the NodeJS application with mocked database. In this example we want to show how we can benefit from Spring Cloud Contract in no time.
* due to those constraints the contracts also represent the stateful situation
  + first request is a POST that causes data to get inserted to the database
  + second request is a GET that returns a list of data with 1 previously inserted element
* the NodeJS application will be started (on port 3000)
* contract tests will be generated via Docker and tests will be executed against the running application
  + the contracts will be taken from /contracts folder.
  + the output of the test execution is available under node\_modules/spring-cloud-contract/output.
* the stubs will be uploaded to Artifactory. You can check them out under <http://localhost:8081/artifactory/libs-release-local/com/example/bookstore/0.0.1.RELEASE/> . The stubs will be here <http://localhost:8081/artifactory/libs-release-local/com/example/bookstore/0.0.1.RELEASE/bookstore-0.0.1.RELEASE-stubs.jar>.

To see how the client side looks like check out the [Section 93.9, “Stub Runner Docker”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#stubrunner-docker) section.

**92. Spring Cloud Contract Verifier Messaging**

Spring Cloud Contract Verifier lets you verify applications that use messaging as a means of communication. All of the integrations shown in this document work with Spring, but you can also create one of your own and use that.

**92.1 Integrations**

You can use one of the following four integration configurations:

* Apache Camel
* Spring Integration
* Spring Cloud Stream
* Spring AMQP

Since we use Spring Boot, if you have added one of these libraries to the classpath, all the messaging configuration is automatically set up.

|  |  |
| --- | --- |
| [Important] | **Important** |
| Remember to put @AutoConfigureMessageVerifier on the base class of your generated tests. Otherwise, messaging part of Spring Cloud Contract Verifier does not work. |
| [Important] | **Important** | |
| If you want to use Spring Cloud Stream, remember to add a dependency on org.springframework.cloud:spring-cloud-stream-test-support, as shown here: | |

**Maven.**

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-stream-test-support</artifactId>

<scope>test</scope>

</dependency>

**Gradle.**

testCompile "org.springframework.cloud:spring-cloud-stream-test-support"

**92.2 Manual Integration Testing**

The main interface used by the tests is org.springframework.cloud.contract.verifier.messaging.MessageVerifier. It defines how to send and receive messages. You can create your own implementation to achieve the same goal.

In a test, you can inject a ContractVerifierMessageExchange to send and receive messages that follow the contract. Then add @AutoConfigureMessageVerifierto your test. Here’s an example:

*@RunWith(SpringTestRunner.class)*

*@SpringBootTest*

*@AutoConfigureMessageVerifier*

**public** **static** **class** MessagingContractTests {

*@Autowired*

**private** MessageVerifier verifier;

...

}

|  |
| --- |
| [Note] |
| If your tests require stubs as well, then @AutoConfigureStubRunner includes the messaging configuration, so you only need the one annotation. |

**92.3 Publisher-Side Test Generation**

Having the input or outputMessage sections in your DSL results in creation of tests on the publisher’s side. By default, JUnit tests are created. However, there is also a possibility to create Spock tests.

There are 3 main scenarios that we should take into consideration:

* Scenario 1: There is no input message that produces an output message. The output message is triggered by a component inside the application (for example, scheduler).
* Scenario 2: The input message triggers an output message.
* Scenario 3: The input message is consumed and there is no output message.

|  |  |
| --- | --- |
| [Important] | **Important** |
| The destination passed to messageFrom or sentTo can have different meanings for different messaging implementations. For **Stream** and **Integration** it is first resolved as a destination of a channel. Then, if there is no such destination it is resolved as a channel name. For **Camel**, that’s a certain component (for example, jms). |

**92.3.1 Scenario 1: No Input Message**

For the given contract:

**Groovy DSL.**

def contractDsl = Contract.make {

label 'some\_label'

input {

triggeredBy('bookReturnedTriggered()')

}

outputMessage {

sentTo('activemq:output')

body('''{ "bookName" : "foo" }''')

headers {

header('BOOK-NAME', 'foo')

messagingContentType(applicationJson())

}

}

}

**YAML.**

label: some\_label

input:

triggeredBy: bookReturnedTriggered

outputMessage:

sentTo: activemq:output

body:

bookName: foo

headers:

BOOK-NAME: foo

contentType: application/json

The following JUnit test is created:

'''

*// when:*

bookReturnedTriggered();

*// then:*

ContractVerifierMessage response = contractVerifierMessaging.receive("activemq:output");

assertThat(response).isNotNull();

assertThat(response.getHeader("BOOK-NAME")).isNotNull();

assertThat(response.getHeader("BOOK-NAME").toString()).isEqualTo("foo");

assertThat(response.getHeader("contentType")).isNotNull();

assertThat(response.getHeader("contentType").toString()).isEqualTo("application/json");

*// and:*

DocumentContext parsedJson = JsonPath.parse(contractVerifierObjectMapper.writeValueAsString(response.getPayload()));

assertThatJson(parsedJson).field("bookName").isEqualTo("foo");

'''

And the following Spock test would be created:

'''

when:

bookReturnedTriggered()

then:

ContractVerifierMessage response = contractVerifierMessaging.receive('activemq:output')

assert response != null

response.getHeader('BOOK-NAME')?.toString() == 'foo'

response.getHeader('contentType')?.toString() == 'application/json'

and:

DocumentContext parsedJson = JsonPath.parse(contractVerifierObjectMapper.writeValueAsString(response.payload))

assertThatJson(parsedJson).field("bookName").isEqualTo("foo")

'''

**92.3.2 Scenario 2: Output Triggered by Input**

For the given contract:

**Groovy DSL.**

def contractDsl = Contract.make {

label 'some\_label'

input {

messageFrom('jms:input')

messageBody([

bookName: 'foo'

])

messageHeaders {

header('sample', 'header')

}

}

outputMessage {

sentTo('jms:output')

body([

bookName: 'foo'

])

headers {

header('BOOK-NAME', 'foo')

}

}

}

**YAML.**

label: some\_label

input:

messageFrom: jms:input

messageBody:

bookName: 'foo'

messageHeaders:

sample: header

outputMessage:

sentTo: jms:output

body:

bookName: foo

headers:

BOOK-NAME: foo

The following JUnit test is created:

'''

*// given:*

ContractVerifierMessage inputMessage = contractVerifierMessaging.create(

"{\\"bookName\\":\\"foo\\"}"

, headers()

.header("sample", "header"));

*// when:*

contractVerifierMessaging.send(inputMessage, "jms:input");

*// then:*

ContractVerifierMessage response = contractVerifierMessaging.receive("jms:output");

assertThat(response).isNotNull();

assertThat(response.getHeader("BOOK-NAME")).isNotNull();

assertThat(response.getHeader("BOOK-NAME").toString()).isEqualTo("foo");

*// and:*

DocumentContext parsedJson = JsonPath.parse(contractVerifierObjectMapper.writeValueAsString(response.getPayload()));

assertThatJson(parsedJson).field("bookName").isEqualTo("foo");

'''

And the following Spock test would be created:

"""\

given:

ContractVerifierMessage inputMessage = contractVerifierMessaging.create(

'''{"bookName":"foo"}''',

['sample': 'header']

)

when:

contractVerifierMessaging.send(inputMessage, 'jms:input')

then:

ContractVerifierMessage response = contractVerifierMessaging.receive('jms:output')

assert response !- null

response.getHeader('BOOK-NAME')?.toString() == 'foo'

and:

DocumentContext parsedJson = JsonPath.parse(contractVerifierObjectMapper.writeValueAsString(response.payload))

assertThatJson(parsedJson).field("bookName").isEqualTo("foo")

"""

**92.3.3 Scenario 3: No Output Message**

For the given contract:

**Groovy DSL.**

def contractDsl = Contract.make {

label 'some\_label'

input {

messageFrom('jms:delete')

messageBody([

bookName: 'foo'

])

messageHeaders {

header('sample', 'header')

}

assertThat('bookWasDeleted()')

}

}

**YAML.**

label: some\_label

input:

messageFrom: jms:delete

messageBody:

bookName: 'foo'

messageHeaders:

sample: header

assertThat: bookWasDeleted()

The following JUnit test is created:

'''

*// given:*

ContractVerifierMessage inputMessage = contractVerifierMessaging.create(

"{\\"bookName\\":\\"foo\\"}"

, headers()

.header("sample", "header"));

*// when:*

contractVerifierMessaging.send(inputMessage, "jms:delete");

*// then:*

bookWasDeleted();

'''

And the following Spock test would be created:

'''

given:

ContractVerifierMessage inputMessage = contractVerifierMessaging.create(

\'\'\'{"bookName":"foo"}\'\'\',

['sample': 'header']

)

when:

contractVerifierMessaging.send(inputMessage, 'jms:delete')

then:

noExceptionThrown()

bookWasDeleted()

'''

**92.4 Consumer Stub Generation**

Unlike the HTTP part, in messaging, we need to publish the Groovy DSL inside the JAR with a stub. Then it is parsed on the consumer side and proper stubbed routes are created.

For more information, see [???](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html) section.

**Maven.**

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-stream-rabbit</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-contract-stub-runner</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-stream-test-support</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Finchley.BUILD-SNAPSHOT</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

**Gradle.**

ext {

contractsDir = file("mappings")

stubsOutputDirRoot = file("${project.buildDir}/production/${project.name}-stubs/")

}

*// Automatically added by plugin:*

*// copyContracts - copies contracts to the output folder from which JAR will be created*

*// verifierStubsJar - JAR with a provided stub suffix*

*// the presented publication is also added by the plugin but you can modify it as you wish*

publishing {

publications {

stubs(MavenPublication) {

artifactId "${project.name}-stubs"

artifact verifierStubsJar

}

}

}

**93. Spring Cloud Contract Stub Runner**

One of the issues that you might encounter while using Spring Cloud Contract Verifier is passing the generated WireMock JSON stubs from the server side to the client side (or to various clients). The same takes place in terms of client-side generation for messaging.

Copying the JSON files and setting the client side for messaging manually is out of the question. That is why we introduced Spring Cloud Contract Stub Runner. It can automatically download and run the stubs for you.

**93.1 Snapshot versions**

Add the additional snapshot repository to your build.gradle file to use snapshot versions, which are automatically uploaded after every successful build:

**Maven.**

<repositories>

<repository>

<id>spring-snapshots</id>

<name>Spring Snapshots</name>

<url>https://repo.spring.io/snapshot</url>

<snapshots>

<enabled>true</enabled>

</snapshots>

</repository>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

<repository>

<id>spring-releases</id>

<name>Spring Releases</name>

<url>https://repo.spring.io/release</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

<pluginRepositories>

<pluginRepository>

<id>spring-snapshots</id>

<name>Spring Snapshots</name>

<url>https://repo.spring.io/snapshot</url>

<snapshots>

<enabled>true</enabled>

</snapshots>

</pluginRepository>

<pluginRepository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</pluginRepository>

<pluginRepository>

<id>spring-releases</id>

<name>Spring Releases</name>

<url>https://repo.spring.io/release</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</pluginRepository>

</pluginRepositories>

**Gradle.**

buildscript {

repositories {

mavenCentral()

mavenLocal()

maven { url "http://repo.spring.io/snapshot" }

maven { url "http://repo.spring.io/milestone" }

maven { url "http://repo.spring.io/release" }

}

**93.2 Publishing Stubs as JARs**

The easiest approach would be to centralize the way stubs are kept. For example, you can keep them as jars in a Maven repository.

|  |
| --- |
| [Tip] |
| For both Maven and Gradle, the setup comes ready to work. However, you can customize it if you want to. |

**Maven.**

*<!-- First disable the default jar setup in the properties section -->*

*<!-- we don't want the verifier to do a jar for us -->*

<spring.cloud.contract.verifier.skip>true</spring.cloud.contract.verifier.skip>

*<!-- Next add the assembly plugin to your build -->*

*<!-- we want the assembly plugin to generate the JAR -->*

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-assembly-plugin</artifactId>

<executions>

<execution>

<id>stub</id>

<phase>prepare-package</phase>

<goals>

<goal>single</goal>

</goals>

<inherited>false</inherited>

<configuration>

<attach>true</attach>

<descriptors>

$../../../../src/assembly/stub.xml

</descriptors>

</configuration>

</execution>

</executions>

</plugin>

*<!-- Finally setup your assembly. Below you can find the contents of src/main/assembly/stub.xml -->*

<assembly

xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3 http://maven.apache.org/xsd/assembly-1.1.3.xsd">

<id>stubs</id>

<formats>

<format>jar</format>

</formats>

<includeBaseDirectory>false</includeBaseDirectory>

<fileSets>

<fileSet>

<directory>src/main/java</directory>

<outputDirectory>/</outputDirectory>

<includes>

<include>\*\*com/example/model/\*.\*</include>

</includes>

</fileSet>

<fileSet>

<directory>${project.build.directory}/classes</directory>

<outputDirectory>/</outputDirectory>

<includes>

<include>\*\*com/example/model/\*.\*</include>

</includes>

</fileSet>

<fileSet>

<directory>${project.build.directory}/snippets/stubs</directory>

<outputDirectory>META-INF/${project.groupId}/${project.artifactId}/${project.version}/mappings</outputDirectory>

<includes>

<include>\*\*/\*</include>

</includes>

</fileSet>

<fileSet>

<directory>$../../../../src/test/resources/contracts</directory>

<outputDirectory>META-INF/${project.groupId}/${project.artifactId}/${project.version}/contracts</outputDirectory>

<includes>

<include>\*\*/\*.groovy</include>

</includes>

</fileSet>

</fileSets>

</assembly>

**Gradle.**

ext {

contractsDir = file("mappings")

stubsOutputDirRoot = file("${project.buildDir}/production/${project.name}-stubs/")

}

*// Automatically added by plugin:*

*// copyContracts - copies contracts to the output folder from which JAR will be created*

*// verifierStubsJar - JAR with a provided stub suffix*

*// the presented publication is also added by the plugin but you can modify it as you wish*

publishing {

publications {

stubs(MavenPublication) {

artifactId "${project.name}-stubs"

artifact verifierStubsJar

}

}

}

**93.3 Stub Runner Core**

Runs stubs for service collaborators. Treating stubs as contracts of services allows to use stub-runner as an implementation of [Consumer Driven Contracts](http://martinfowler.com/articles/consumerDrivenContracts.html).

Stub Runner allows you to automatically download the stubs of the provided dependencies (or pick those from the classpath), start WireMock servers for them and feed them with proper stub definitions. For messaging, special stub routes are defined.

**93.3.1 Retrieving stubs**

You can pick the following options of acquiring stubs

* Aether based solution that downloads JARs with stubs from Artifactory / Nexus
* Classpath scanning solution that searches classpath via pattern to retrieve stubs
* Write your own implementation of the org.springframework.cloud.contract.stubrunner.StubDownloaderBuilder for full customization

The latter example is described in the [Custom Stub Runner](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html) section.

**Stub downloading**

You can control the stub downloading via the stubsMode switch. It picks value from the StubRunnerProperties.StubsMode enum. You can use the following options

* StubRunnerProperties.StubsMode.CLASSPATH (default value) - will pick stubs from the classpath
* StubRunnerProperties.StubsMode.LOCAL - will pick stubs from a local storage (e.g. .m2)
* StubRunnerProperties.StubsMode.REMOTE - will pick stubs from a remote location

Example:

@AutoConfigureStubRunner(repositoryRoot="http://foo.bar", ids = "com.example:beer-api-producer:+:stubs:8095", stubsMode = StubRunnerProperties.StubsMode.LOCAL)

**Classpath scanning**

If you set the stubsMode property to StubRunnerProperties.StubsMode.CLASSPATH (or set nothing since CLASSPATH is the default value) then classpath will get scanned. Let’s look at the following example:

@AutoConfigureStubRunner(ids = {

"com.example:beer-api-producer:+:stubs:8095",

"com.example.foo:bar:1.0.0:superstubs:8096"

})

If you’ve added the dependencies to your classpath

**Maven.**

<dependency>

<groupId>com.example</groupId>

<artifactId>beer-api-producer-restdocs</artifactId>

<classifier>stubs</classifier>

<version>0.0.1-SNAPSHOT</version>

<scope>test</scope>

<exclusions>

<exclusion>

<groupId>\*</groupId>

<artifactId>\*</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>com.example.foo</groupId>

<artifactId>bar</artifactId>

<classifier>superstubs</classifier>

<version>1.0.0</version>

<scope>test</scope>

<exclusions>

<exclusion>

<groupId>\*</groupId>

<artifactId>\*</artifactId>

</exclusion>

</exclusions>

</dependency>

**Gradle.**

testCompile("com.example:beer-api-producer-restdocs:0.0.1-SNAPSHOT:stubs") {

transitive = false

}

testCompile("com.example.foo:bar:1.0.0:superstubs") {

transitive = false

}

Then the following locations on your classpath will get scanned. For com.example:beer-api-producer-restdocs

* /META-INF/com.example/beer-api-producer-restdocs/**\*/**.\*
* /contracts/com.example/beer-api-producer-restdocs/**\*/**.\*
* /mappings/com.example/beer-api-producer-restdocs/**\*/**.\*

and com.example.foo:bar

* /META-INF/com.example.foo/bar/**\*/**.\*
* /contracts/com.example.foo/bar/**\*/**.\*
* /mappings/com.example.foo/bar/**\*/**.\*

|  |
| --- |
| [Tip] |
| As you can see you have to explicitly provide the group and artifact ids when packaging the producer stubs. |

The producer would setup the contracts like this:

└── src

└── **test**

└── resources

└── contracts

   └── com.example

      └── beer-api-producer-restdocs

      └── nested

      └── contract3.groovy

To achieve proper stub packaging.

Or using the [Maven assembly plugin](https://github.com/spring-cloud-samples/spring-cloud-contract-samples/blob/2.0.x/producer_with_restdocs/pom.xml) or [Gradle Jar](https://github.com/spring-cloud-samples/spring-cloud-contract-samples/blob/2.0.x/producer_with_restdocs/build.gradle) task you have to create the following structure in your stubs jar.

└── META-INF

└── com.example

└── beer-api-producer-restdocs

└── 2.0.0

├── contracts

│   └── nested

   │ └── contract2.groovy

   └── mappings

   └── mapping.json

By maintaining this structure classpath gets scanned and you can profit from the messaging / HTTP stubs without the need to download artifacts.

**93.3.2 Running stubs**

**Limitations**

|  |  |
| --- | --- |
| [Important] | **Important** |
| There might be a problem with StubRunner shutting down ports between tests. You might have a situation in which you get port conflicts. As long as you use the same context across tests everything works fine. But when the context are different (e.g. different stubs or different profiles) then you have to either use @DirtiesContext to shut down the stub servers, or else run them on different ports per test. |

**Running using main app**

You can set the following options to the main class:

-c, --classifier Suffix **for** the jar containing stubs (e.

g. 'stubs' **if** the stub jar would

have a 'stubs' classifier **for** stubs:

foobar-stubs ). Defaults to 'stubs'

(**default**: stubs)

--maxPort, --maxp <Integer> Maximum port value to be assigned to

the WireMock instance. Defaults to

15000 (**default**: 15000)

--minPort, --minp <Integer> Minimum port value to be assigned to

the WireMock instance. Defaults to

10000 (**default**: 10000)

-p, --password Password to user when connecting to

repository

--phost, --proxyHost Proxy host to use **for** repository

requests

--pport, --proxyPort [Integer] Proxy port to use **for** repository

requests

-r, --root Location of a Jar containing server

where you keep your stubs (e.g. http:

*//nexus.*

net/content/repositories/repository)

-s, --stubs Comma separated list of Ivy

representation of jars with stubs.

Eg. groupid:artifactid1,groupid2:

artifactid2:classifier

--sm, --stubsMode Stubs mode to be used. Acceptable values

[CLASSPATH, LOCAL, REMOTE]

-u, --username Username to user when connecting to

repository

**HTTP Stubs**

Stubs are defined in JSON documents, whose syntax is defined in [WireMock documentation](http://wiremock.org/stubbing.html)

Example:

{

"request": {

"method": "GET",

"url": "/ping"

},

"response": {

"status": 200,

"body": "pong",

"headers": {

"Content-Type": "text/plain"

}

}

}

**Viewing registered mappings**

Every stubbed collaborator exposes list of defined mappings under \_\_/admin/ endpoint.

You can also use the mappingsOutputFolder property to dump the mappings to files. For annotation based approach it would look like this

@AutoConfigureStubRunner(ids="a.b.c:loanIssuance,a.b.c:fraudDetectionServer",

mappingsOutputFolder = "target/outputmappings/")

and for the JUnit approach like this:

*@ClassRule* *@Shared* StubRunnerRule rule = **new** StubRunnerRule()

.repoRoot("http://some\_url")

.downloadStub("a.b.c", "loanIssuance")

.downloadStub("a.b.c:fraudDetectionServer")

.withMappingsOutputFolder("target/outputmappings")

Then if you check out the folder target/outputmappings you would see the following structure

.

├── fraudDetectionServer\_13705

└── loanIssuance\_12255

That means that there were two stubs registered. fraudDetectionServer was registered at port 13705 and loanIssuance at port 12255. If we take a look at one of the files we would see (for WireMock) mappings available for the given server:

[**{**

"id" : "f9152eb9-bf77-4c38-8289-90be7d10d0d7"**,**

"request" : **{**

"url" : "/name"**,**

"method" : "GET"

**},**

"response" : **{**

"status" : 200**,**

"body" : "fraudDetectionServer"

**},**

"uuid" : "f9152eb9-bf77-4c38-8289-90be7d10d0d7"

**},**

...

**]**

**Messaging Stubs**

Depending on the provided Stub Runner dependency and the DSL the messaging routes are automatically set up.

**93.4 Stub Runner JUnit Rule**

Stub Runner comes with a JUnit rule thanks to which you can very easily download and run stubs for given group and artifact id:

*@ClassRule* **public** **static** StubRunnerRule rule = **new** StubRunnerRule()

.repoRoot(repoRoot())

.downloadStub("org.springframework.cloud.contract.verifier.stubs", "loanIssuance")

.downloadStub("org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer");

After that rule gets executed Stub Runner connects to your Maven repository and for the given list of dependencies tries to:

* download them
* cache them locally
* unzip them to a temporary folder
* start a WireMock server for each Maven dependency on a random port from the provided range of ports / provided port
* feed the WireMock server with all JSON files that are valid WireMock definitions
* can also send messages (remember to pass an implementation of MessageVerifier interface)

Stub Runner uses [Eclipse Aether](https://wiki.eclipse.org/Aether) mechanism to download the Maven dependencies. Check their [docs](https://wiki.eclipse.org/Aether) for more information.

Since the StubRunnerRule implements the StubFinder it allows you to find the started stubs:

**package** org.springframework.cloud.contract.stubrunner;

**import** java.net.URL;

**import** java.util.Collection;

**import** java.util.Map;

**import** org.springframework.cloud.contract.spec.Contract;

**public** **interface** StubFinder **extends** StubTrigger {

**/\*\***

**\* For the given groupId and artifactId tries to find the matching**

**\* URL of the running stub.**

**\***

**\* @param groupId - might be null. In that case a search only via artifactId takes place**

**\* @return URL of a running stub or throws exception if not found**

**\*/**

URL findStubUrl(String groupId, String artifactId) **throws** StubNotFoundException;

**/\*\***

**\* For the given Ivy notation {@code [groupId]:artifactId:[version]:[classifier]} tries to**

**\* find the matching URL of the running stub. You can also pass only {@code artifactId}.**

**\***

**\* @param ivyNotation - Ivy representation of the Maven artifact**

**\* @return URL of a running stub or throws exception if not found**

**\*/**

URL findStubUrl(String ivyNotation) **throws** StubNotFoundException;

**/\*\***

**\* Returns all running stubs**

**\*/**

RunningStubs findAllRunningStubs();

**/\*\***

**\* Returns the list of Contracts**

**\*/**

Map<StubConfiguration, Collection<Contract>> getContracts();

}

Example of usage in Spock tests:

*@ClassRule* *@Shared* StubRunnerRule rule = **new** StubRunnerRule()

.stubsMode(StubRunnerProperties.StubsMode.REMOTE)

.repoRoot(StubRunnerRuleSpec.getResource("/m2repo/repository").toURI().toString())

.downloadStub("org.springframework.cloud.contract.verifier.stubs", "loanIssuance")

.downloadStub("org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer")

.withMappingsOutputFolder("target/outputmappingsforrule")

def 'should start WireMock servers'() {

expect: 'WireMocks are running'

rule.findStubUrl('org.springframework.cloud.contract.verifier.stubs', 'loanIssuance') != null

rule.findStubUrl('loanIssuance') != null

rule.findStubUrl('loanIssuance') == rule.findStubUrl('org.springframework.cloud.contract.verifier.stubs', 'loanIssuance')

rule.findStubUrl('org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer') != null

and:

rule.findAllRunningStubs().isPresent('loanIssuance')

rule.findAllRunningStubs().isPresent('org.springframework.cloud.contract.verifier.stubs', 'fraudDetectionServer')

rule.findAllRunningStubs().isPresent('org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer')

and: 'Stubs were registered'

"${rule.findStubUrl('loanIssuance').toString()}/name".toURL().text == 'loanIssuance'

"${rule.findStubUrl('fraudDetectionServer').toString()}/name".toURL().text == 'fraudDetectionServer'

}

def 'should output mappings to output folder'() {

when:

def url = rule.findStubUrl('fraudDetectionServer')

then:

**new** File("target/outputmappingsforrule", "fraudDetectionServer\_${url.port}").exists()

}

Example of usage in JUnit tests:

*@Test*

**public** **void** should\_start\_wiremock\_servers() **throws** Exception {

*// expect: 'WireMocks are running'*

then(rule.findStubUrl("org.springframework.cloud.contract.verifier.stubs", "loanIssuance")).isNotNull();

then(rule.findStubUrl("loanIssuance")).isNotNull();

then(rule.findStubUrl("loanIssuance")).isEqualTo(rule.findStubUrl("org.springframework.cloud.contract.verifier.stubs", "loanIssuance"));

then(rule.findStubUrl("org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer")).isNotNull();

*// and:*

then(rule.findAllRunningStubs().isPresent("loanIssuance")).isTrue();

then(rule.findAllRunningStubs().isPresent("org.springframework.cloud.contract.verifier.stubs", "fraudDetectionServer")).isTrue();

then(rule.findAllRunningStubs().isPresent("org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer")).isTrue();

*// and: 'Stubs were registered'*

then(httpGet(rule.findStubUrl("loanIssuance").toString() + "/name")).isEqualTo("loanIssuance");

then(httpGet(rule.findStubUrl("fraudDetectionServer").toString() + "/name")).isEqualTo("fraudDetectionServer");

}

Check the **Common properties for JUnit and Spring** for more information on how to apply global configuration of Stub Runner.

|  |  |
| --- | --- |
| [Important] | **Important** |
| To use the JUnit rule together with messaging you have to provide an implementation of the MessageVerifier interface to the rule builder (e.g. rule.messageVerifier(new MyMessageVerifier())). If you don’t do this then whenever you try to send a message an exception will be thrown. |

**93.4.1 Maven settings**

The stub downloader honors Maven settings for a different local repository folder. Authentication details for repositories and profiles are currently not taken into account, so you need to specify it using the properties mentioned above.

**93.4.2 Providing fixed ports**

You can also run your stubs on fixed ports. You can do it in two different ways. One is to pass it in the properties, and the other via fluent API of JUnit rule.

**93.4.3 Fluent API**

When using the StubRunnerRule you can add a stub to download and then pass the port for the last downloaded stub.

*@ClassRule* **public** **static** StubRunnerRule rule = **new** StubRunnerRule()

.repoRoot(repoRoot())

.downloadStub("org.springframework.cloud.contract.verifier.stubs", "loanIssuance")

.withPort(12345)

.downloadStub("org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer:12346");

You can see that for this example the following test is valid:

then(rule.findStubUrl("loanIssuance")).isEqualTo(URI.create("http://localhost:12345").toURL());

then(rule.findStubUrl("fraudDetectionServer")).isEqualTo(URI.create("http://localhost:12346").toURL());

**93.4.4 Stub Runner with Spring**

Sets up Spring configuration of the Stub Runner project.

By providing a list of stubs inside your configuration file the Stub Runner automatically downloads and registers in WireMock the selected stubs.

If you want to find the URL of your stubbed dependency you can autowire the StubFinder interface and use its methods as presented below:

*@ContextConfiguration(classes = Config, loader = SpringBootContextLoader)*

*@SpringBootTest(properties = [" stubrunner.cloud.enabled=false",*

*'foo=${stubrunner.runningstubs.fraudDetectionServer.port}',*

*'fooWithGroup=${stubrunner.runningstubs.org.springframework.cloud.contract.verifier.stubs.fraudDetectionServer.port}'])*

*@AutoConfigureStubRunner(mappingsOutputFolder = "target/outputmappings/")*

*@DirtiesContext*

*@ActiveProfiles("test")*

**class** StubRunnerConfigurationSpec **extends** Specification {

*@Autowired* StubFinder stubFinder

*@Autowired* Environment environment

*@StubRunnerPort("fraudDetectionServer")* **int** fraudDetectionServerPort

*@StubRunnerPort("org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer")* **int** fraudDetectionServerPortWithGroupId

*@Value('${foo}')* Integer foo

*@BeforeClass*

*@AfterClass*

**void** setupProps() {

System.clearProperty("stubrunner.repository.root")

System.clearProperty("stubrunner.classifier")

}

def 'should start WireMock servers'() {

expect: 'WireMocks are running'

stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs', 'loanIssuance') != null

stubFinder.findStubUrl('loanIssuance') != null

stubFinder.findStubUrl('loanIssuance') == stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs', 'loanIssuance')

stubFinder.findStubUrl('loanIssuance') == stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs:loanIssuance')

stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs:loanIssuance:0.0.1-SNAPSHOT') == stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs:loanIssuance:0.0.1-SNAPSHOT:stubs')

stubFinder.findStubUrl('org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer') != null

and:

stubFinder.findAllRunningStubs().isPresent('loanIssuance')

stubFinder.findAllRunningStubs().isPresent('org.springframework.cloud.contract.verifier.stubs', 'fraudDetectionServer')

stubFinder.findAllRunningStubs().isPresent('org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer')

and: 'Stubs were registered'

"${stubFinder.findStubUrl('loanIssuance').toString()}/name".toURL().text == 'loanIssuance'

"${stubFinder.findStubUrl('fraudDetectionServer').toString()}/name".toURL().text == 'fraudDetectionServer'

}

def 'should throw an exception when stub is not found'() {

when:

stubFinder.findStubUrl('nonExistingService')

then:

thrown(StubNotFoundException)

when:

stubFinder.findStubUrl('nonExistingGroupId', 'nonExistingArtifactId')

then:

thrown(StubNotFoundException)

}

def 'should register started servers as environment variables'() {

expect:

environment.getProperty("stubrunner.runningstubs.loanIssuance.port") != null

stubFinder.findAllRunningStubs().getPort("loanIssuance") == (environment.getProperty("stubrunner.runningstubs.loanIssuance.port") as Integer)

and:

environment.getProperty("stubrunner.runningstubs.fraudDetectionServer.port") != null

stubFinder.findAllRunningStubs().getPort("fraudDetectionServer") == (environment.getProperty("stubrunner.runningstubs.fraudDetectionServer.port") as Integer)

and:

environment.getProperty("stubrunner.runningstubs.fraudDetectionServer.port") != null

stubFinder.findAllRunningStubs().getPort("fraudDetectionServer") == (environment.getProperty("stubrunner.runningstubs.org.springframework.cloud.contract.verifier.stubs.fraudDetectionServer.port") as Integer)

}

def 'should be able to interpolate a running stub in the passed test property'() {

given:

**int** fraudPort = stubFinder.findAllRunningStubs().getPort("fraudDetectionServer")

expect:

fraudPort > 0

environment.getProperty("foo", Integer) == fraudPort

environment.getProperty("fooWithGroup", Integer) == fraudPort

foo == fraudPort

}

*@Issue("#573")*

def 'should be able to retrieve the port of a running stub via an annotation'() {

given:

**int** fraudPort = stubFinder.findAllRunningStubs().getPort("fraudDetectionServer")

expect:

fraudPort > 0

fraudDetectionServerPort == fraudPort

fraudDetectionServerPortWithGroupId == fraudPort

}

def 'should dump all mappings to a file'() {

when:

def url = stubFinder.findStubUrl("fraudDetectionServer")

then:

**new** File("target/outputmappings/", "fraudDetectionServer\_${url.port}").exists()

}

*@Configuration*

*@EnableAutoConfiguration*

**static** **class** Config {}

}

for the following configuration file:

stubrunner:

repositoryRoot: classpath:m2repo/repository/

ids:

- org.springframework.cloud.contract.verifier.stubs:loanIssuance

- org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer

- org.springframework.cloud.contract.verifier.stubs:bootService

stubs-mode: remote

Instead of using the properties you can also use the properties inside the @AutoConfigureStubRunner. Below you can find an example of achieving the same result by setting values on the annotation.

@AutoConfigureStubRunner(

ids = ["org.springframework.cloud.contract.verifier.stubs:loanIssuance",

"org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer",

"org.springframework.cloud.contract.verifier.stubs:bootService"],

stubsMode = StubRunnerProperties.StubsMode.REMOTE,

repositoryRoot = "classpath:m2repo/repository/")

Stub Runner Spring registers environment variables in the following manner for every registered WireMock server. Example for Stub Runner ids com.example:foo, com.example:bar.

* stubrunner.runningstubs.foo.port
* stubrunner.runningstubs.com.example.foo.port
* stubrunner.runningstubs.bar.port
* stubrunner.runningstubs.com.example.bar.port

Which you can reference in your code.

You can also use the @StubRunnerPort annotation to inject the port of a running stub. Value of the annotation can be the groupid:artifactid or just the artifactid. Example for Stub Runner ids com.example:foo, com.example:bar.

*@StubRunnerPort("foo")*

**int** fooPort;

*@StubRunnerPort("com.example:bar")*

**int** barPort;

**93.5 Stub Runner Spring Cloud**

Stub Runner can integrate with Spring Cloud.

For real life examples you can check the

* [producer app sample](https://github.com/spring-cloud-samples/spring-cloud-contract-samples/tree/2.0.x/producer)
* [consumer app sample](https://github.com/spring-cloud-samples/spring-cloud-contract-samples/tree/2.0.x/consumer_with_discovery)

**93.5.1 Stubbing Service Discovery**

The most important feature of Stub Runner Spring Cloud is the fact that it’s stubbing

* DiscoveryClient
* Ribbon ServerList

that means that regardless of the fact whether you’re using Zookeeper, Consul, Eureka or anything else, you don’t need that in your tests. We’re starting WireMock instances of your dependencies and we’re telling your application whenever you’re using Feign, load balanced RestTemplate or DiscoveryClient directly, to call those stubbed servers instead of calling the real Service Discovery tool.

For example this test will pass

def 'should make service discovery work'() {

expect: 'WireMocks are running'

"${stubFinder.findStubUrl('loanIssuance').toString()}/name".toURL().text == 'loanIssuance'

"${stubFinder.findStubUrl('fraudDetectionServer').toString()}/name".toURL().text == 'fraudDetectionServer'

and: 'Stubs can be reached via load service discovery'

restTemplate.getForObject('http://loanIssuance/name', String) == 'loanIssuance'

restTemplate.getForObject('http://someNameThatShouldMapFraudDetectionServer/name', String) == 'fraudDetectionServer'

}

for the following configuration file

stubrunner:

idsToServiceIds:

ivyNotation: someValueInsideYourCode

fraudDetectionServer: someNameThatShouldMapFraudDetectionServer

**Test profiles and service discovery**

In your integration tests you typically don’t want to call neither a discovery service (e.g. Eureka) or Config Server. That’s why you create an additional test configuration in which you want to disable these features.

Due to certain limitations of [spring-cloud-commons](https://github.com/spring-cloud/spring-cloud-commons/issues/156) to achieve this you have disable these properties via a static block like presented below (example for Eureka)

*//Hack to work around https://github.com/spring-cloud/spring-cloud-commons/issues/156*

**static** {

System.setProperty("eureka.client.enabled", "false");

System.setProperty("spring.cloud.config.failFast", "false");

}

**93.5.2 Additional Configuration**

You can match the artifactId of the stub with the name of your app by using the stubrunner.idsToServiceIds: map. You can disable Stub Runner Ribbon support by providing: stubrunner.cloud.ribbon.enabled equal to false You can disable Stub Runner support by providing: stubrunner.cloud.enabled equal to false

|  |
| --- |
| [Tip] |
| By default all service discovery will be stubbed. That means that regardless of the fact if you have an existing DiscoveryClient its results will be ignored. However, if you want to reuse it, just set stubrunner.cloud.delegate.enabled to true and then your existing DiscoveryClient results will be merged with the stubbed ones. |

The default Maven configuration used by Stub Runner can be tweaked either via the following system properties or environment variables

* maven.repo.local - path to the custom maven local repository location
* org.apache.maven.user-settings - path to custom maven user settings location
* org.apache.maven.global-settings - path to maven global settings location

**93.6 Stub Runner Boot Application**

Spring Cloud Contract Stub Runner Boot is a Spring Boot application that exposes REST endpoints to trigger the messaging labels and to access started WireMock servers.

One of the use-cases is to run some smoke (end to end) tests on a deployed application. You can check out the [Spring Cloud Pipelines](https://github.com/spring-cloud/spring-cloud-pipelines) project for more information.

**93.6.1 How to use it?**

**Stub Runner Server**

Just add the

compile "org.springframework.cloud:spring-cloud-starter-stub-runner"

Annotate a class with @EnableStubRunnerServer, build a fat-jar and you’re ready to go!

For the properties check the **Stub Runner Spring** section.

**Stub Runner Server Fat Jar**

You can download a standalone JAR from Maven (for example, for version 1.2.3.RELEASE), as follows:

$ wget -O stub-runner.jar 'https://search.maven.org/remote\_content?g=org.springframework.cloud&a=spring-cloud-contract-stub-runner-boot&v=1.2.3.RELEASE'

$ java -jar stub-runner.jar --stubrunner.ids=... --stubrunner.repositoryRoot=...

**Spring Cloud CLI**

Starting from 1.4.0.RELEASE version of the [Spring Cloud CLI](https://cloud.spring.io/spring-cloud-cli) project you can start Stub Runner Boot by executing spring cloud stubrunner.

In order to pass the configuration just create a stubrunner.yml file in the current working directory or a subdirectory called config or in ~/.spring-cloud. The file could look like this (example for running stubs installed locally)

**stubrunner.yml.**

stubrunner:

stubsMode: LOCAL

ids:

- com.example:beer-api-producer:+:9876

and then just call spring cloud stubrunner from your terminal window to start the Stub Runner server. It will be available at port 8750.

**93.6.2 Endpoints**

**HTTP**

* GET /stubs - returns a list of all running stubs in ivy:integer notation
* GET /stubs/{ivy} - returns a port for the given ivy notation (when calling the endpoint ivy can also be artifactId only)

**Messaging**

For Messaging

* GET /triggers - returns a list of all running labels in ivy : [ label1, label2 …​] notation
* POST /triggers/{label} - executes a trigger with label
* POST /triggers/{ivy}/{label} - executes a trigger with label for the given ivy notation (when calling the endpoint ivy can also be artifactId only)

**93.6.3 Example**

*@ContextConfiguration(classes = StubRunnerBoot, loader = SpringBootContextLoader)*

*@SpringBootTest(properties = "spring.cloud.zookeeper.enabled=false")*

*@ActiveProfiles("test")*

**class** StubRunnerBootSpec **extends** Specification {

*@Autowired* StubRunning stubRunning

def setup() {

RestAssuredMockMvc.standaloneSetup(**new** HttpStubsController(stubRunning),

**new** TriggerController(stubRunning))

}

def 'should return a list of running stub servers in "full ivy:port" notation'() {

when:

String response = RestAssuredMockMvc.get('/stubs').body.asString()

then:

def root = **new** JsonSlurper().parseText(response)

root.'org.springframework.cloud.contract.verifier.stubs:bootService:0.0.1-SNAPSHOT:stubs' **instanceof** Integer

}

def 'should return a port on which a [#stubId] stub is running'() {

when:

def response = RestAssuredMockMvc.get("/stubs/${stubId}")

then:

response.statusCode == 200

Integer.valueOf(response.body.asString()) > 0

where:

stubId << ['org.springframework.cloud.contract.verifier.stubs:bootService:+:stubs',

'org.springframework.cloud.contract.verifier.stubs:bootService:0.0.1-SNAPSHOT:stubs',

'org.springframework.cloud.contract.verifier.stubs:bootService:+',

'org.springframework.cloud.contract.verifier.stubs:bootService',

'bootService']

}

def 'should return 404 when missing stub was called'() {

when:

def response = RestAssuredMockMvc.get("/stubs/a:b:c:d")

then:

response.statusCode == 404

}

def 'should return a list of messaging labels that can be triggered when version and classifier are passed'() {

when:

String response = RestAssuredMockMvc.get('/triggers').body.asString()

then:

def root = **new** JsonSlurper().parseText(response)

root.'org.springframework.cloud.contract.verifier.stubs:bootService:0.0.1-SNAPSHOT:stubs'?.containsAll(["delete\_book","return\_book\_1","return\_book\_2"])

}

def 'should trigger a messaging label'() {

given:

StubRunning stubRunning = Mock()

RestAssuredMockMvc.standaloneSetup(**new** HttpStubsController(stubRunning), **new** TriggerController(stubRunning))

when:

def response = RestAssuredMockMvc.post("/triggers/delete\_book")

then:

response.statusCode == 200

and:

1 \* stubRunning.trigger('delete\_book')

}

def 'should trigger a messaging label for a stub with [#stubId] ivy notation'() {

given:

StubRunning stubRunning = Mock()

RestAssuredMockMvc.standaloneSetup(**new** HttpStubsController(stubRunning), **new** TriggerController(stubRunning))

when:

def response = RestAssuredMockMvc.post("/triggers/$stubId/delete\_book")

then:

response.statusCode == 200

and:

1 \* stubRunning.trigger(stubId, 'delete\_book')

where:

stubId << ['org.springframework.cloud.contract.verifier.stubs:bootService:stubs', 'org.springframework.cloud.contract.verifier.stubs:bootService', 'bootService']

}

def 'should throw exception when trigger is missing'() {

when:

RestAssuredMockMvc.post("/triggers/missing\_label")

then:

Exception e = thrown(Exception)

e.message.contains("Exception occurred while trying to return [missing\_label] label.")

e.message.contains("Available labels are")

e.message.contains("org.springframework.cloud.contract.verifier.stubs:loanIssuance:0.0.1-SNAPSHOT:stubs=[]")

e.message.contains("org.springframework.cloud.contract.verifier.stubs:bootService:0.0.1-SNAPSHOT:stubs=")

}

}

**93.6.4 Stub Runner Boot with Service Discovery**

One of the possibilities of using Stub Runner Boot is to use it as a feed of stubs for "smoke-tests". What does it mean? Let’s assume that you don’t want to deploy 50 microservice to a test environment in order to check if your application is working fine. You’ve already executed a suite of tests during the build process but you would also like to ensure that the packaging of your application is fine. What you can do is to deploy your application to an environment, start it and run a couple of tests on it to see if it’s working fine. We can call those tests smoke-tests since their idea is to check only a handful of testing scenarios.

The problem with this approach is such that if you’re doing microservices most likely you’re using a service discovery tool. Stub Runner Boot allows you to solve this issue by starting the required stubs and register them in a service discovery tool. Let’s take a look at an example of such a setup with Eureka. Let’s assume that Eureka was already running.

*@SpringBootApplication*

*@EnableStubRunnerServer*

*@EnableEurekaClient*

*@AutoConfigureStubRunner*

**public** **class** StubRunnerBootEurekaExample {

**public** **static** **void** main(String[] args) {

SpringApplication.run(StubRunnerBootEurekaExample.**class**, args);

}

}

As you can see we want to start a Stub Runner Boot server @EnableStubRunnerServer, enable Eureka client @EnableEurekaClient and we want to have the stub runner feature turned on @AutoConfigureStubRunner.

Now let’s assume that we want to start this application so that the stubs get automatically registered. We can do it by running the app java -jar ${SYSTEM\_PROPS} stub-runner-boot-eureka-example.jar where ${SYSTEM\_PROPS} would contain the following list of properties

-Dstubrunner.repositoryRoot=http://repo.spring.io/snapshots (1)

-Dstubrunner.cloud.stubbed.discovery.enabled=false (2)

-Dstubrunner.ids=org.springframework.cloud.contract.verifier.stubs:loanIssuance,org.springframework.cloud.contract.verifier.stubs:fraudDetectionServer,org.springframework.cloud.contract.verifier.stubs:bootService (3)

-Dstubrunner.idsToServiceIds.fraudDetectionServer=someNameThatShouldMapFraudDetectionServer (4)

(1) - we tell Stub Runner where all the stubs reside

(2) - we don't want the default behaviour where the discovery service is stubbed. That's why the stub registration will be picked

(3) - we provide a list of stubs to download

(4) - we provide a list of artifactId to serviceId mapping

That way your deployed application can send requests to started WireMock servers via the service discovery. Most likely points 1-3 could be set by default in application.yml cause they are not likely to change. That way you can provide only the list of stubs to download whenever you start the Stub Runner Boot.

**93.7 Stubs Per Consumer**

There are cases in which 2 consumers of the same endpoint want to have 2 different responses.

|  |
| --- |
| [Tip] |
| This approach also allows you to immediately know which consumer is using which part of your API. You can remove part of a response that your API produces and you can see which of your autogenerated tests fails. If none fails then you can safely delete that part of the response cause nobody is using it. |

Let’s look at the following example for contract defined for the producer called producer. There are 2 consumers: foo-consumer and bar-consumer.

**Consumer foo-service**

request {

url '/foo'

method GET()

}

response {

status OK()

body(

foo: "foo"

}

}

**Consumer bar-service**

request {

url '/foo'

method GET()

}

response {

status OK()

body(

bar: "bar"

}

}

You can’t produce for the same request 2 different responses. That’s why you can properly package the contracts and then profit from the stubsPerConsumer feature.

On the producer side the consumers can have a folder that contains contracts related only to them. By setting the stubrunner.stubs-per-consumer flag to true we no longer register all stubs but only those that correspond to the consumer application’s name. In other words we’ll scan the path of every stub and if it contains the subfolder with name of the consumer in the path only then will it get registered.

On the foo producer side the contracts would look like this

.

└── contracts

├── bar-consumer

│   ├── bookReturnedForBar.groovy

│   └── shouldCallBar.groovy

└── foo-consumer

├── bookReturnedForFoo.groovy

└── shouldCallFoo.groovy

Being the bar-consumer consumer you can either set the spring.application.name or the stubrunner.consumer-name to bar-consumer Or set the test as follows:

*@ContextConfiguration(classes = Config, loader = SpringBootContextLoader)*

*@SpringBootTest(properties = ["spring.application.name=bar-consumer"])*

*@AutoConfigureStubRunner(ids = "org.springframework.cloud.contract.verifier.stubs:producerWithMultipleConsumers",*

*repositoryRoot = "classpath:m2repo/repository/",*

*stubsMode = StubRunnerProperties.StubsMode.REMOTE,*

*stubsPerConsumer = true)*

*@DirtiesContext*

**class** StubRunnerStubsPerConsumerSpec **extends** Specification {

...

}

Then only the stubs registered under a path that contains the bar-consumer in its name (i.e. those from thesrc/test/resources/contracts/bar-consumer/some/contracts/…​ folder) will be allowed to be referenced.

Or set the consumer name explicitly

*@ContextConfiguration(classes = Config, loader = SpringBootContextLoader)*

*@SpringBootTest*

*@AutoConfigureStubRunner(ids = "org.springframework.cloud.contract.verifier.stubs:producerWithMultipleConsumers",*

*repositoryRoot = "classpath:m2repo/repository/",*

*consumerName = "foo-consumer",*

*stubsMode = StubRunnerProperties.StubsMode.REMOTE,*

*stubsPerConsumer = true)*

*@DirtiesContext*

**class** StubRunnerStubsPerConsumerWithConsumerNameSpec **extends** Specification {

...

}

Then only the stubs registered under a path that contains the foo-consumer in its name (i.e. those from thesrc/test/resources/contracts/foo-consumer/some/contracts/…​ folder) will be allowed to be referenced.

You can check out [issue 224](https://github.com/spring-cloud/spring-cloud-contract/issues/224) for more information about the reasons behind this change.

**93.8 Common**

This section briefly describes common properties, including:

* [Section 93.8.1, “Common Properties for JUnit and Spring”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#common-properties-junit-spring)
* [Section 93.8.2, “Stub Runner Stubs IDs”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#stub-runner-stub-ids)

**93.8.1 Common Properties for JUnit and Spring**

You can set repetitive properties by using system properties or Spring configuration properties. Here are their names with their default values:

| **Property name** | **Default value** | **Description** |
| --- | --- | --- |
| stubrunner.minPort | 10000 | Minimum value of a port for a started WireMock with stubs. |
| stubrunner.maxPort | 15000 | Maximum value of a port for a started WireMock with stubs. |
| stubrunner.repositoryRoot |  | Maven repo URL. If blank, then call the local maven repo. |
| stubrunner.classifier | stubs | Default classifier for the stub artifacts. |
| stubrunner.stubsMode | CLASSPATH | The way you want to fetch and register the stubs |
| stubrunner.ids |  | Array of Ivy notation stubs to download. |
| stubrunner.username |  | Optional username to access the tool that stores the JARs with stubs. |
| stubrunner.password |  | Optional password to access the tool that stores the JARs with stubs. |
| stubrunner.stubsPerConsumer | false | Set to true if you want to use different stubs for each consumer instead of registering all stubs for every consumer. |
| stubrunner.consumerName |  | If you want to use a stub for each consumer and want to override the consumer name just change this value. |

**93.8.2 Stub Runner Stubs IDs**

You can provide the stubs to download via the stubrunner.ids system property. They follow this pattern:

groupId:artifactId:version:classifier:port

Note that version, classifier and port are optional.

* If you do not provide the port, a random one will be picked.
* If you do not provide the classifier, the default is used. (Note that you can pass an empty classifier this way: groupId:artifactId:version:).
* If you do not provide the version, then the + will be passed and the latest one is downloaded.

port means the port of the WireMock server.

|  |  |
| --- | --- |
| [Important] | **Important** |
| Starting with version 1.0.4, you can provide a range of versions that you would like the Stub Runner to take into consideration. You can read more about the[Aether versioning ranges here](https://wiki.eclipse.org/Aether/New_and_Noteworthy#Version_Ranges). |

**93.9 Stub Runner Docker**

We’re publishing a spring-cloud/spring-cloud-contract-stub-runner Docker image that will start the standalone version of Stub Runner.

If you want to learn more about the basics of Maven, artifact ids, group ids, classifiers and Artifact Managers, just click here [Section 91.6, “Docker Project”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#docker-project).

**93.9.1 How to use it**

Just execute the docker image. You can pass any of the [Section 93.8.1, “Common Properties for JUnit and Spring”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#common-properties-junit-spring) as environment variables. The convention is that all the letters should be upper case. The camel case notation should and the dot (.) should be separated via underscore (\_). E.g. the stubrunner.repositoryRootproperty should be represented as a STUBRUNNER\_REPOSITORY\_ROOT environment variable.

**93.9.2 Example of client side usage in a non JVM project**

We’d like to use the stubs created in this [Section 91.6.4, “Server side (nodejs)”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#docker-server-side) step. Let’s assume that we want to run the stubs on port 9876. The NodeJS code is available here:

$ git clone https://github.com/spring-cloud-samples/spring-cloud-contract-nodejs

$ **cd** bookstore

Let’s run the Stub Runner Boot application with the stubs.

*# Provide the Spring Cloud Contract Docker version*

$ SC\_CONTRACT\_DOCKER\_VERSION="..."

*# The IP at which the app is running and Docker container can reach it*

$ APP\_IP="192.168.0.100"

*# Spring Cloud Contract Stub Runner properties*

$ STUBRUNNER\_PORT="8083"

*# Stub coordinates 'groupId:artifactId:version:classifier:port'*

$ STUBRUNNER\_IDS="com.example:bookstore:0.0.1.RELEASE:stubs:9876"

$ STUBRUNNER\_REPOSITORY\_ROOT="http://${APP\_IP}:8081/artifactory/libs-release-local"

*# Run the docker with Stub Runner Boot*

$ docker run --rm -e "STUBRUNNER\_IDS=${STUBRUNNER\_IDS}" -e "STUBRUNNER\_REPOSITORY\_ROOT=${STUBRUNNER\_REPOSITORY\_ROOT}" -e "STUBRUNNER\_STUBS\_MODE=REMOTE" -p "${STUBRUNNER\_PORT}:${STUBRUNNER\_PORT}" -p "9876:9876" springcloud/spring-cloud-contract-stub-runner:"${SC\_CONTRACT\_DOCKER\_VERSION}"

What’s happening is that

* a standalone Stub Runner application got started
* it downloaded the stub with coordinates com.example:bookstore:0.0.1.RELEASE:stubs on port 9876
* it got downloaded from Artifactory running at <http://192.168.0.100:8081/artifactory/libs-release-local>
* after a while Stub Runner will be running on port 8083
* and the stubs will be running at port 9876

On the server side we built a stateful stub. Let’s use curl to assert that the stubs are setup properly.

*# let's execute the first request (no response is returned)*

$ curl -H "Content-Type:application/json" -X POST --data '{ "title" : "Title", "genre" : "Genre", "description" : "Description", "author" : "Author", "publisher" : "Publisher", "pages" : 100, "image\_url" : "https://d213dhlpdb53mu.cloudfront.net/assets/pivotal-square-logo-41418bd391196c3022f3cd9f3959b3f6d7764c47873d858583384e759c7db435.svg", "buy\_url" : "https://pivotal.io" }' http://localhost:9876/api/books

*# Now time for the second request*

$ curl -X GET http://localhost:9876/api/books

*# You will receive contents of the JSON*

|  |  |
| --- | --- |
| [Important] | **Important** |
| If you want use the stubs that you have built locally, on your host, then you should pass the environment variable -e STUBRUNNER\_STUBS\_MODE=LOCAL and mount the volume of your local m2 -v "${HOME}/.m2/:/root/.m2:ro" |

**94. Stub Runner for Messaging**

Stub Runner can run the published stubs in memory. It can integrate with the following frameworks:

* Spring Integration
* Spring Cloud Stream
* Spring AMQP

It also provides entry points to integrate with any other solution on the market.

|  |  |
| --- | --- |
| [Important] | **Important** |
| If you have multiple frameworks on the classpath Stub Runner will need to define which one should be used. Let’s assume that you have both AMQP, Spring Cloud Stream and Spring Integration on the classpath. Then you need to set stubrunner.stream.enabled=false and stubrunner.integration.enabled=false. That way the only remaining framework is Spring AMQP. |

**94.1 Stub triggering**

To trigger a message, use the StubTrigger interface:

**package** org.springframework.cloud.contract.stubrunner;

**import** java.util.Collection;

**import** java.util.Map;

**public** **interface** StubTrigger {

**/\*\***

**\* Triggers an event by a given label for a given {@code groupid:artifactid} notation. You can use only {@code artifactId} too.**

**\***

**\* Feature related to messaging.**

**\***

**\* @return true - if managed to run a trigger**

**\*/**

**boolean** trigger(String ivyNotation, String labelName);

**/\*\***

**\* Triggers an event by a given label.**

**\***

**\* Feature related to messaging.**

**\***

**\* @return true - if managed to run a trigger**

**\*/**

**boolean** trigger(String labelName);

**/\*\***

**\* Triggers all possible events.**

**\***

**\* Feature related to messaging.**

**\***

**\* @return true - if managed to run a trigger**

**\*/**

**boolean** trigger();

**/\*\***

**\* Returns a mapping of ivy notation of a dependency to all the labels it has.**

**\***

**\* Feature related to messaging.**

**\*/**

Map<String, Collection<String>> labels();

}

For convenience, the StubFinder interface extends StubTrigger, so you only need one or the other in your tests.

StubTrigger gives you the following options to trigger a message:

* [Section 94.1.1, “Trigger by Label”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#trigger-label)
* [Section 94.1.2, “Trigger by Group and Artifact Ids”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#trigger-group-artifact-ids)
* [Section 94.1.3, “Trigger by Artifact Ids”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#trigger-artifact-ids)
* [Section 94.1.4, “Trigger All Messages”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#trigger-all-messages)

**94.1.1 Trigger by Label**

stubFinder.trigger('return\_book\_1')

**94.1.2 Trigger by Group and Artifact Ids**

stubFinder.trigger('org.springframework.cloud.contract.verifier.stubs:streamService', 'return\_book\_1')

**94.1.3 Trigger by Artifact Ids**

stubFinder.trigger('streamService', 'return\_book\_1')

**94.1.4 Trigger All Messages**

stubFinder.trigger()

**94.2 Stub Runner Integration**

Spring Cloud Contract Verifier Stub Runner’s messaging module gives you an easy way to integrate with Spring Integration. For the provided artifacts, it automatically downloads the stubs and registers the required routes.

**94.2.1 Adding the Runner to the Project**

You can have both Spring Integration and Spring Cloud Contract Stub Runner on the classpath. Remember to annotate your test class with @AutoConfigureStubRunner.

**94.2.2 Disabling the functionality**

If you need to disable this functionality, set the stubrunner.integration.enabled=false property.

Assume that you have the following Maven repository with deployed stubs for the integrationService application:

└── .m2

└── repository

└── io

└── codearte

└── accurest

└── stubs

└── integrationService

├── 0.0.1-SNAPSHOT

│   ├── integrationService-0.0.1-SNAPSHOT.pom

│   ├── integrationService-0.0.1-SNAPSHOT-stubs.jar

│   └── maven-metadata-local.xml

└── maven-metadata-local.xml

Further assume the stubs contain the following structure:

├── META-INF

│   └── MANIFEST.MF

└── repository

├── accurest

│   ├── bookDeleted.groovy

│   ├── bookReturned1.groovy

│   └── bookReturned2.groovy

└── mappings

Consider the following contracts (numbered **1**):

Contract.make {

label 'return\_book\_1'

input {

triggeredBy('bookReturnedTriggered()')

}

outputMessage {

sentTo('output')

body('''{ "bookName" : "foo" }''')

headers {

header('BOOK-NAME', 'foo')

}

}

}

Now consider **2**:

Contract.make {

label 'return\_book\_2'

input {

messageFrom('input')

messageBody([

bookName: 'foo'

])

messageHeaders {

header('sample', 'header')

}

}

outputMessage {

sentTo('output')

body([

bookName: 'foo'

])

headers {

header('BOOK-NAME', 'foo')

}

}

}

and the following Spring Integration Route:

<?xml version="1.0" encoding="UTF-8"?>

<beans:beans xmlns="http://www.springframework.org/schema/integration"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:beans="http://www.springframework.org/schema/beans"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/integration

http://www.springframework.org/schema/integration/spring-integration.xsd">

*<!-- REQUIRED FOR TESTING -->*

<bridge input-channel="output"

output-channel="outputTest"/>

<channel id="outputTest">

<queue/>

</channel>

</beans:beans>

These examples lend themselves to three scenarios:

* [the section called “Scenario 1 (no input message)”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#integration-scenario-1)
* [???](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html)
* [the section called “Scenario 3 (input with no output)”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#integration-scenario-3)

**Scenario 1 (no input message)**

To trigger a message via the return\_book\_1 label, use the StubTigger interface, as follows:

stubFinder.trigger('return\_book\_1')

To listen to the output of the message sent to output:

Message<?> receivedMessage = messaging.receive('outputTest')

The received message would pass the following assertions:

receivedMessage != null

assertJsons(receivedMessage.payload)

receivedMessage.headers.get('BOOK-NAME') == 'foo'

**Scenario 2 (output triggered by input)**

Since the route is set for you, you can send a message to the output destination:

messaging.send(**new** BookReturned('foo'), [sample: 'header'], 'input')

To listen to the output of the message sent to output:

Message<?> receivedMessage = messaging.receive('outputTest')

The received message passes the following assertions:

receivedMessage != null

assertJsons(receivedMessage.payload)

receivedMessage.headers.get('BOOK-NAME') == 'foo'

**Scenario 3 (input with no output)**

Since the route is set for you, you can send a message to the input destination:

messaging.send(**new** BookReturned('foo'), [sample: 'header'], 'delete')

**94.3 Stub Runner Stream**

Spring Cloud Contract Verifier Stub Runner’s messaging module gives you an easy way to integrate with Spring Stream. For the provided artifacts, it automatically downloads the stubs and registers the required routes.

|  |
| --- |
| [Warning] |
| If Stub Runner’s integration with Stream the messageFrom or sentTo Strings are resolved first as a destination of a channel and no such destination exists, the destination is resolved as a channel name. |
| [Important] | **Important** | |
| If you want to use Spring Cloud Stream remember, to add a dependency on org.springframework.cloud:spring-cloud-stream-test-support. | |

**Maven.**

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-stream-test-support</artifactId>

<scope>test</scope>

</dependency>

**Gradle.**

testCompile "org.springframework.cloud:spring-cloud-stream-test-support"

**94.3.1 Adding the Runner to the Project**

You can have both Spring Cloud Stream and Spring Cloud Contract Stub Runner on the classpath. Remember to annotate your test class with @AutoConfigureStubRunner.

**94.3.2 Disabling the functionality**

If you need to disable this functionality, set the stubrunner.stream.enabled=false property.

Assume that you have the following Maven repository with a deployed stubs for the streamService application:

└── .m2

└── repository

└── io

└── codearte

└── accurest

└── stubs

└── streamService

├── 0.0.1-SNAPSHOT

│   ├── streamService-0.0.1-SNAPSHOT.pom

│   ├── streamService-0.0.1-SNAPSHOT-stubs.jar

│   └── maven-metadata-local.xml

└── maven-metadata-local.xml

Further assume the stubs contain the following structure:

├── META-INF

│   └── MANIFEST.MF

└── repository

├── accurest

│   ├── bookDeleted.groovy

│   ├── bookReturned1.groovy

│   └── bookReturned2.groovy

└── mappings

Consider the following contracts (numbered **1**):

Contract.make {

label 'return\_book\_1'

input { triggeredBy('bookReturnedTriggered()') }

outputMessage {

sentTo('returnBook')

body('''{ "bookName" : "foo" }''')

headers { header('BOOK-NAME', 'foo') }

}

}

Now consider **2**:

Contract.make {

label 'return\_book\_2'

input {

messageFrom('bookStorage')

messageBody([

bookName: 'foo'

])

messageHeaders { header('sample', 'header') }

}

outputMessage {

sentTo('returnBook')

body([

bookName: 'foo'

])

headers { header('BOOK-NAME', 'foo') }

}

}

Now consider the following Spring configuration:

stubrunner.repositoryRoot: classpath:m2repo/repository/

stubrunner.ids: org.springframework.cloud.contract.verifier.stubs:streamService:0.0.1-SNAPSHOT:stubs

stubrunner.stubs-mode: remote

spring:

cloud:

stream:

bindings:

output:

destination: returnBook

input:

destination: bookStorage

server:

port: 0

debug: **true**

These examples lend themselves to three scenarios:

* [the section called “Scenario 1 (no input message)”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#stream-scenario-1)
* [the section called “Scenario 2 (output triggered by input)”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#stream-scenario-2)
* [the section called “Scenario 3 (input with no output)”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#stream-scenario-3)

**Scenario 1 (no input message)**

To trigger a message via the return\_book\_1 label, use the StubTrigger interface as follows:

stubFinder.trigger('return\_book\_1')

To listen to the output of the message sent to a channel whose destination is returnBook:

Message<?> receivedMessage = messaging.receive('returnBook')

The received message passes the following assertions:

receivedMessage != null

assertJsons(receivedMessage.payload)

receivedMessage.headers.get('BOOK-NAME') == 'foo'

**Scenario 2 (output triggered by input)**

Since the route is set for you, you can send a message to the bookStorage destination:

messaging.send(**new** BookReturned('foo'), [sample: 'header'], 'bookStorage')

To listen to the output of the message sent to returnBook:

Message<?> receivedMessage = messaging.receive('returnBook')

The received message passes the following assertions:

receivedMessage != null

assertJsons(receivedMessage.payload)

receivedMessage.headers.get('BOOK-NAME') == 'foo'

**Scenario 3 (input with no output)**

Since the route is set for you, you can send a message to the output destination:

messaging.send(**new** BookReturned('foo'), [sample: 'header'], 'delete')

**94.4 Stub Runner Spring AMQP**

Spring Cloud Contract Verifier Stub Runner’s messaging module provides an easy way to integrate with Spring AMQP’s Rabbit Template. For the provided artifacts, it automatically downloads the stubs and registers the required routes.

The integration tries to work standalone (that is, without interaction with a running RabbitMQ message broker). It expects a RabbitTemplate on the application context and uses it as a spring boot test named @SpyBean. As a result, it can use the mockito spy functionality to verify and inspect messages sent by the application.

On the message consumer side, the stub runner considers all @RabbitListener annotated endpoints and all SimpleMessageListenerContainer objects on the application context.

As messages are usually sent to exchanges in AMQP, the message contract contains the exchange name as the destination. Message listeners on the other side are bound to queues. Bindings connect an exchange to a queue. If message contracts are triggered, the Spring AMQP stub runner integration looks for bindings on the application context that match this exchange. Then it collects the queues from the Spring exchanges and tries to find message listeners bound to these queues. The message is triggered for all matching message listeners.

**94.4.1 Adding the Runner to the Project**

You can have both Spring AMQP and Spring Cloud Contract Stub Runner on the classpath and set the property stubrunner.amqp.enabled=true. Remember to annotate your test class with @AutoConfigureStubRunner.

|  |  |
| --- | --- |
| [Important] | **Important** |
| If you already have Stream and Integration on the classpath, you need to disable them explicitly by setting the stubrunner.stream.enabled=false andstubrunner.integration.enabled=false properties. |

Assume that you have the following Maven repository with a deployed stubs for the spring-cloud-contract-amqp-test application.

└── .m2

└── repository

└── com

└── example

└── spring-cloud-contract-amqp-**test**

├── 0.4.0-SNAPSHOT

│   ├── spring-cloud-contract-amqp-**test**-0.4.0-SNAPSHOT.pom

│   ├── spring-cloud-contract-amqp-**test**-0.4.0-SNAPSHOT-stubs.jar

│   └── maven-metadata-local.xml

└── maven-metadata-local.xml

Further assume that the stubs contain the following structure:

├── META-INF

│   └── MANIFEST.MF

└── contracts

└── shouldProduceValidPersonData.groovy

Consider the following contract:

Contract.make {

*// Human readable description*

description 'Should produce valid person data'

*// Label by means of which the output message can be triggered*

label 'contract-test.person.created.event'

*// input to the contract*

input {

*// the contract will be triggered by a method*

triggeredBy('createPerson()')

}

*// output message of the contract*

outputMessage {

*// destination to which the output message will be sent*

sentTo 'contract-test.exchange'

headers {

header('contentType': 'application/json')

header('\_\_TypeId\_\_': 'org.springframework.cloud.contract.stubrunner.messaging.amqp.Person')

}

*// the body of the output message*

body ([

id: $(consumer(9), producer(regex("[0-9]+"))),

name: "me"

])

}

}

Now consider the following Spring configuration:

stubrunner:

repositoryRoot: classpath:m2repo/repository/

ids: org.springframework.cloud.contract.verifier.stubs.amqp:spring-cloud-contract-amqp-test:0.4.0-SNAPSHOT:stubs

stubs-mode: remote

amqp:

enabled: **true**

server:

port: 0

**Triggering the message**

To trigger a message using the contract above, use the StubTrigger interface as follows:

stubTrigger.trigger("contract-test.person.created.event")

The message has a destination of contract-test.exchange, so the Spring AMQP stub runner integration looks for bindings related to this exchange.

*@Bean*

**public** Binding binding() {

**return** BindingBuilder.bind(**new** Queue("test.queue")).to(**new** DirectExchange("contract-test.exchange")).with("#");

}

The binding definition binds the queue test.queue. As a result, the following listener definition is matched and invoked with the contract message.

*@Bean*

**public** SimpleMessageListenerContainer simpleMessageListenerContainer(ConnectionFactory connectionFactory,

MessageListenerAdapter listenerAdapter) {

SimpleMessageListenerContainer container = **new** SimpleMessageListenerContainer();

container.setConnectionFactory(connectionFactory);

container.setQueueNames("test.queue");

container.setMessageListener(listenerAdapter);

**return** container;

}

Also, the following annotated listener matches and is invoked:

*@RabbitListener(bindings = @QueueBinding(*

*value = @Queue(value = "test.queue"),*

*exchange = @Exchange(value = "contract-test.exchange", ignoreDeclarationExceptions = "true")))*

**public** **void** handlePerson(Person person) {

**this**.person = person;

}

|  |
| --- |
| [Note] |
| The message is directly handed over to the onMessage method of the MessageListener associated with the matching SimpleMessageListenerContainer. |

**Spring AMQP Test Configuration**

In order to avoid Spring AMQP trying to connect to a running broker during our tests configure a mock ConnectionFactory.

To disable the mocked ConnectionFactory, set the following property: stubrunner.amqp.mockConnection=false

stubrunner:

amqp:

mockConnection: **false**

**95. Contract DSL**

Spring Cloud Contract supports out of the box 2 types of DSL. One written in Groovy and one written in YAML.

If you decide to write the contract in Groovy, do not be alarmed if you have not used Groovy before. Knowledge of the language is not really needed, as the Contract DSL uses only a tiny subset of it (only literals, method calls and closures). Also, the DSL is statically typed, to make it programmer-readable without any knowledge of the DSL itself.

|  |  |
| --- | --- |
| [Important] | **Important** |
| Remember that, inside the Groovy contract file, you have to provide the fully qualified name to the Contract class and make static imports, such asorg.springframework.cloud.spec.Contract.make { …​ }. You can also provide an import to the Contract class: import org.springframework.cloud.spec.Contract and then call Contract.make { …​ }. |
| [Tip] |
| Spring Cloud Contract supports defining multiple contracts in a single file. | |

The following is a complete example of a Groovy contract definition:

org.springframework.cloud.contract.spec.Contract.make {

request {

method 'PUT'

url '/api/12'

headers {

header 'Content-Type': 'application/vnd.org.springframework.cloud.contract.verifier.twitter-places-analyzer.v1+json'

}

body '''\

[{

"created\_at": "Sat Jul 26 09:38:57 +0000 2014",

"id": 492967299297845248,

"id\_str": "492967299297845248",

"text": "Gonna see you at Warsaw",

"place":

{

"attributes":{},

"bounding\_box":

{

"coordinates":

[[

[-77.119759,38.791645],

[-76.909393,38.791645],

[-76.909393,38.995548],

[-77.119759,38.995548]

]],

"type":"Polygon"

},

"country":"United States",

"country\_code":"US",

"full\_name":"Washington, DC",

"id":"01fbe706f872cb32",

"name":"Washington",

"place\_type":"city",

"url": "http://api.twitter.com/1/geo/id/01fbe706f872cb32.json"

}

}]

'''

}

response {

status OK()

}

}

The following is a complete example of a YAML contract definition:

description: Some description

name: some name

priority: 8

ignored: true

request:

url: /foo

queryParameters:

a: b

b: c

method: PUT

headers:

foo: bar

fooReq: baz

body:

foo: bar

matchers:

body:

- path: $.foo

type: by\_regex

value: bar

headers:

- key: foo

regex: bar

response:

status: 200

headers:

foo2: bar

foo3: foo33

fooRes: baz

body:

foo2: bar

foo3: baz

nullValue: null

matchers:

body:

- path: $.foo2

type: by\_regex

value: bar

- path: $.foo3

type: by\_command

value: executeMe($it)

- path: $.nullValue

type: by\_null

value: null

headers:

- key: foo2

regex: bar

- key: foo3

command: andMeToo($it)

|  |
| --- |
| [Tip] |
| You can compile contracts to stubs mapping using standalone maven command:mvn org.springframework.cloud:spring-cloud-contract-maven-plugin:convert |

**95.1 Limitations**

|  |
| --- |
| [Warning] |
| Spring Cloud Contract Verifier does not properly support XML. Please use JSON or help us implement this feature. |
| [Warning] |
| The support for verifying the size of JSON arrays is experimental. If you want to turn it on, please set the value of the following system property to true:spring.cloud.contract.verifier.assert.size. By default, this feature is set to false. You can also provide the assertJsonSize property in the plugin configuration. | |

|  |
| --- |
| [Warning] |
| Because JSON structure can have any form, it can be impossible to parse it properly when using the Groovy DSL and the value(consumer(…​), producer(…​)) notation in GString. That is why you should use the Groovy Map notation. |

**95.2 Common Top-Level elements**

The following sections describe the most common top-level elements:

* [Section 95.2.1, “Description”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#contract-dsl-description)
* [Section 95.2.2, “Name”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#contract-dsl-name)
* [Section 95.2.3, “Ignoring Contracts”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#contract-dsl-ignoring-contracts)
* [Section 95.2.4, “Passing Values from Files”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#contract-dsl-passing-values-from-files)
* [Section 95.2.5, “HTTP Top-Level Elements”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#contract-dsl-http-top-level-elements)

**95.2.1 Description**

You can add a description to your contract. The description is arbitrary text. The following code shows an example:

**Groovy DSL.**

org.springframework.cloud.contract.spec.Contract.make {

description('''

given:

An input

when:

Sth happens

then:

Output

''')

}

**YAML.**

description: Some description

name: some name

priority: 8

ignored: true

request:

url: /foo

queryParameters:

a: b

b: c

method: PUT

headers:

foo: bar

fooReq: baz

body:

foo: bar

matchers:

body:

- path: $.foo

type: by\_regex

value: bar

headers:

- key: foo

regex: bar

response:

status: 200

headers:

foo2: bar

foo3: foo33

fooRes: baz

body:

foo2: bar

foo3: baz

nullValue: null

matchers:

body:

- path: $.foo2

type: by\_regex

value: bar

- path: $.foo3

type: by\_command

value: executeMe($it)

- path: $.nullValue

type: by\_null

value: null

headers:

- key: foo2

regex: bar

- key: foo3

command: andMeToo($it)

**95.2.2 Name**

You can provide a name for your contract. Assume that you provided the following name: should register a user. If you do so, the name of the autogenerated test isvalidate\_should\_register\_a\_user. Also, the name of the stub in a WireMock stub is should\_register\_a\_user.json.

|  |  |
| --- | --- |
| [Important] | **Important** |
| You must ensure that the name does not contain any characters that make the generated test not compile. Also, remember that, if you provide the same name for multiple contracts, your autogenerated tests fail to compile and your generated stubs override each other. |

**Groovy DSL.**

org.springframework.cloud.contract.spec.Contract.make {

name("some\_special\_name")

}

**YAML.**

name: some name

**95.2.3 Ignoring Contracts**

If you want to ignore a contract, you can either set a value of ignored contracts in the plugin configuration or set the ignored property on the contract itself:

**Groovy DSL.**

org.springframework.cloud.contract.spec.Contract.make {

ignored()

}

**YAML.**

ignored: true

**95.2.4 Passing Values from Files**

Starting with version 1.2.0, you can pass values from files. Assume that you have the following resources in our project.

└── src

    └── **test**

       └── resources

          └── contracts

    ├── readFromFile.groovy

    ├── request.json

    └── response.json

Further assume that your contract is as follows:

**Groovy DSL.**

**import** org.springframework.cloud.contract.spec.Contract

Contract.make {

request {

method('PUT')

headers {

contentType(applicationJson())

}

body(file("request.json"))

url("/1")

}

response {

status OK()

body(file("response.json"))

headers {

contentType(textPlain())

}

}

}

**YAML.**

request:

method: GET

url: /foo

bodyFromFile: request.json

response:

status: 200

bodyFromFile: response.json

Further assume that the JSON files is as follows:

**request.json**

**{** "status" : "REQUEST" **}**

**response.json**

**{** "status" : "RESPONSE" **}**

When test or stub generation takes place, the contents of the file is passed to the body of a request or a response. The name of the file needs to be a file with location relative to the folder in which the contract lays.

**95.2.5 HTTP Top-Level Elements**

The following methods can be called in the top-level closure of a contract definition. request and response are mandatory. priority is optional.

**Groovy DSL.**

org.springframework.cloud.contract.spec.Contract.make {

*// Definition of HTTP request part of the contract*

*// (this can be a valid request or invalid depending*

*// on type of contract being specified).*

request {

*//...*

}

*// Definition of HTTP response part of the contract*

*// (a service implementing this contract should respond*

*// with following response after receiving request*

*// specified in "request" part above).*

response {

*//...*

}

*// Contract priority, which can be used for overriding*

*// contracts (1 is highest). Priority is optional.*

priority 1

}

**YAML.**

priority: 8

request:

...

response:

...

|  |  |
| --- | --- |
| [Important] | **Important** |
| If you want to make your contract have a **higher** value of priority you need to pass a **lower** number to the priority tag / method. E.g. priority with value 5 has **higher** priority than priority with value 10. |

**95.3 Request**

The HTTP protocol requires only **method and url** to be specified in a request. The same information is mandatory in request definition of the Contract.

**Groovy DSL.**

org.springframework.cloud.contract.spec.Contract.make {

request {

*// HTTP request method (GET/POST/PUT/DELETE).*

method 'GET'

*// Path component of request URL is specified as follows.*

urlPath('/users')

}

response {

*//...*

}

}

**YAML.**

method: PUT

url: /foo

It is possible to specify an absolute rather than relative url, but using urlPath is the recommended way, as doing so makes the tests **host-independent**.

**Groovy DSL.**

org.springframework.cloud.contract.spec.Contract.make {

request {

method 'GET'

*// Specifying `url` and `urlPath` in one contract is illegal.*

url('http://localhost:8888/users')

}

response {

*//...*

}

}

**YAML.**

request:

method: PUT

urlPath: /foo

request may contain **query parameters**.

**Groovy DSL.**

org.springframework.cloud.contract.spec.Contract.make {

request {

*//...*

urlPath('/users') {

*// Each parameter is specified in form*

*// `'paramName' : paramValue` where parameter value*

*// may be a simple literal or one of matcher functions,*

*// all of which are used in this example.*

queryParameters {

*// If a simple literal is used as value*

*// default matcher function is used (equalTo)*

parameter 'limit': 100

*// `equalTo` function simply compares passed value*

*// using identity operator (==).*

parameter 'filter': equalTo("email")

*// `containing` function matches strings*

*// that contains passed substring.*

parameter 'gender': value(consumer(containing("[mf]")), producer('mf'))

*// `matching` function tests parameter*

*// against passed regular expression.*

parameter 'offset': value(consumer(matching("[0-9]+")), producer(123))

*// `notMatching` functions tests if parameter*

*// does not match passed regular expression.*

parameter 'loginStartsWith': value(consumer(notMatching(".{0,2}")), producer(3))

}

}

*//...*

}

response {

*//...*

}

}

**YAML.**

request:

...

queryParameters:

a: b

b: c

headers:

foo: bar

fooReq: baz

cookies:

foo: bar

fooReq: baz

body:

foo: bar

matchers:

body:

- path: $.foo

type: by\_regex

value: bar

headers:

- key: foo

regex: bar

response:

status: 200

headers:

foo2: bar

foo3: foo33

fooRes: baz

body:

foo2: bar

foo3: baz

nullValue: null

matchers:

body:

- path: $.foo2

type: by\_regex

value: bar

- path: $.foo3

type: by\_command

value: executeMe($it)

- path: $.nullValue

type: by\_null

value: null

headers:

- key: foo2

regex: bar

- key: foo3

command: andMeToo($it)

cookies:

- key: foo2

regex: bar

- key: foo3

predefined:

request may contain additional **request headers**, as shown in the following example:

**Groovy DSL.**

org.springframework.cloud.contract.spec.Contract.make {

request {

*//...*

*// Each header is added in form `'Header-Name' : 'Header-Value'`.*

*// there are also some helper methods*

headers {

header 'key': 'value'

contentType(applicationJson())

}

*//...*

}

response {

*//...*

}

}

**YAML.**

request:

...

headers:

foo: bar

fooReq: baz

request may contain additional **request cookies**, as shown in the following example:

**Groovy DSL.**

org.springframework.cloud.contract.spec.Contract.make {

request {

*//...*

*// Each Cookies is added in form `'Cookie-Key' : 'Cookie-Value'`.*

*// there are also some helper methods*

cookies {

cookie 'key': 'value'

cookie('another\_key', 'another\_value')

}

*//...*

}

response {

*//...*

}

}

**YAML.**

request:

...

cookies:

foo: bar

fooReq: baz

request may contain a **request body**:

**Groovy DSL.**

org.springframework.cloud.contract.spec.Contract.make {

request {

*//...*

*// Currently only JSON format of request body is supported.*

*// Format will be determined from a header or body's content.*

body '''{ "login" : "john", "name": "John The Contract" }'''

}

response {

*//...*

}

}

**YAML.**

request:

...

body:

foo: bar

request may contain **multipart** elements. To include multipart elements, use the multipart method/section, as shown in the following examples

**Groovy DSL.**

org.springframework.cloud.contract.spec.Contract contractDsl = org.springframework.cloud.contract.spec.Contract.make {

request {

method "PUT"

url "/multipart"

headers {

contentType('multipart/form-data;boundary=AaB03x')

}

multipart(

*// key (parameter name), value (parameter value) pair*

formParameter: $(c(regex('".+"')), p('"formParameterValue"')),

someBooleanParameter: $(c(regex(anyBoolean())), p('true')),

*// a named parameter (e.g. with `file` name) that represents file with*

*// `name` and `content`. You can also call `named("fileName", "fileContent")`*

file: named(

*// name of the file*

name: $(c(regex(nonEmpty())), p('filename.csv')),

*// content of the file*

content: $(c(regex(nonEmpty())), p('file content')),

*// content type for the part*

contentType: $(c(regex(nonEmpty())), p('application/json')))

)

}

response {

status OK()

}

}

org.springframework.cloud.contract.spec.Contract contractDsl = org.springframework.cloud.contract.spec.Contract.make {

request {

method "PUT"

url "/multipart"

headers {

contentType('multipart/form-data;boundary=AaB03x')

}

multipart(

file: named(

name: value(stub(regex('.+')), test('file')),

content: value(stub(regex('.+')), test([100, 117, 100, 97] as **byte**[]))

)

)

}

response {

status 200

}

}

**YAML.**

request:

method: PUT

url: /multipart

headers:

Content-Type: multipart/form-data;boundary=AaB03x

multipart:

params:

# key (parameter name), value (parameter value) pair

formParameter: '"formParameterValue"'

someBooleanParameter: true

named:

- paramName: file

fileName: filename.csv

fileContent: file content

matchers:

multipart:

params:

- key: formParameter

regex: ".+"

- key: someBooleanParameter

predefined: any\_boolean

named:

- paramName: file

fileName:

predefined: non\_empty

fileContent:

predefined: non\_empty

response:

status: 200

In the preceding example, we define parameters in either of two ways:

**Groovy DSL**

* Directly, by using the map notation, where the value can be a dynamic property (such as formParameter: $(consumer(…​), producer(…​))).
* By using the named(…​) method that lets you set a named parameter. A named parameter can set a name and content. You can call it either via a method with two arguments, such as named("fileName", "fileContent"), or via a map notation, such as named(name: "fileName", content: "fileContent").

**YAML**

* The multipart parameters are set via multipart.params section
* The named parameters (the fileName and fileContent for a given parameter name) can be set via the multipart.named section. That section contains the paramName (name of the parameter), fileName (name of the file), fileContent (content of the file) fields
* The dynamic bits can be set via the matchers.multipart section
  + for parameters use the params section that can accept regex or a predefined regular expression
  + for named params use the named section where first you define the parameter name via paramName and then you can pass the parametrization of either fileName or fileContent via regex or a predefined regular expression

From this contract, the generated test is as follows:

*// given:*

MockMvcRequestSpecification request = given()

.header("Content-Type", "multipart/form-data;boundary=AaB03x")

.param("formParameter", "\"formParameterValue\"")

.param("someBooleanParameter", "true")

.multiPart("file", "filename.csv", "file content".getBytes());

*// when:*

ResponseOptions response = given().spec(request)

.put("/multipart");

*// then:*

assertThat(response.statusCode()).isEqualTo(200);

The WireMock stub is as follows:

'''

**{**

"request" : **{**

"url" : "/multipart"**,**

"method" : "PUT"**,**

"headers" : **{**

"Content-Type" : **{**

"matches" : "multipart/form-data;boundary=AaB03x.\*"

**}**

**},**

"bodyPatterns" : **[** **{**

"matches" : ".\*--(.\*)\\r\\nContent-Disposition: form-data; name=\\"formParameter\\"\\r\\n(Content-Type: .\*\\r\\n)?(Content-Transfer-Encoding: .\*\\r\\n)?(Content-Length: \\\\d+\\r\\n)?\\r\\n\\".+\\"\\r\\n--\\\\1.\*"

**},** **{**

"matches" : ".\*--(.\*)\\r\\nContent-Disposition: form-data; name=\\"someBooleanParameter\\"\\r\\n(Content-Type: .\*\\r\\n)?(Content-Transfer-Encoding: .\*\\r\\n)?(Content-Length: \\\\d+\\r\\n)?\\r\\n(true|false)\\r\\n--\\\\1.\*"

**},** **{**

"matches" : ".\*--(.\*)\\r\\nContent-Disposition: form-data; name=\\"file\\"; filename=\\"[\\\\S\\\\s]+\\"\\r\\n(Content-Type: .\*\\r\\n)?(Content-Transfer-Encoding: .\*\\r\\n)?(Content-Length: \\\\d+\\r\\n)?\\r\\n[\\\\S\\\\s]+\\r\\n--\\\\1.\*"

**}** **]**

**},**

"response" : **{**

"status" : 200**,**

"transformers" : **[** "response-template"**,** "foo-transformer" **]**

**}**

**}**

'''

**95.4 Response**

The response must contain an **HTTP status code** and may contain other information. The following code shows an example:

**Groovy DSL.**

org.springframework.cloud.contract.spec.Contract.make {

request {

*//...*

}

response {

*// Status code sent by the server*

*// in response to request specified above.*

status OK()

}

}

**YAML.**

response:

...

status: 200

Besides status, the response may contain **headers**, **cookies** and a **body**, both of which are specified the same way as in the request (see the previous paragraph).

|  |
| --- |
| [Tip] |
| Via the Groovy DSL you can reference the org.springframework.cloud.contract.spec.internal.HttpStatus methods to provide a meaningful status instead of a digit. E.g. you can call OK() for a status 200 or BAD\_REQUEST() for 400. |

**95.5 Dynamic properties**

The contract can contain some dynamic properties: timestamps, IDs, and so on. You do not want to force the consumers to stub their clocks to always return the same value of time so that it gets matched by the stub.

For Groovy DSL you can provide the dynamic parts in your contracts in two ways: pass them directly in the body or set them in a separate section called bodyMatchers.

|  |
| --- |
| [Note] |
| Before 2.0.0 these were set using testMatchers and stubMatchers, check out the [migration guide](https://github.com/spring-cloud/spring-cloud-contract/wiki/Spring-Cloud-Contract-2.0-Migration-Guide) for more information. |

For YAML you can only use the matchers section.

**95.5.1 Dynamic properties inside the body**

|  |  |
| --- | --- |
| [Important] | **Important** |
| This section is valid only for Groovy DSL. Check out the [Section 95.5.7, “Dynamic Properties in the Matchers Sections”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#contract-matchers) section for YAML examples of a similar feature. |

You can set the properties inside the body either with the value method or, if you use the Groovy map notation, with $(). The following example shows how to set dynamic properties with the value method:

value(consumer(...), producer(...))

value(c(...), p(...))

value(stub(...), test(...))

value(client(...), server(...))

The following example shows how to set dynamic properties with $():

$(consumer(...), producer(...))

$(c(...), p(...))

$(stub(...), test(...))

$(client(...), server(...))

Both approaches work equally well. stub and client methods are aliases over the consumer method. Subsequent sections take a closer look at what you can do with those values.

**95.5.2 Regular expressions**

|  |  |
| --- | --- |
| [Important] | **Important** |
| This section is valid only for Groovy DSL. Check out the [Section 95.5.7, “Dynamic Properties in the Matchers Sections”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#contract-matchers) section for YAML examples of a similar feature. |

You can use regular expressions to write your requests in Contract DSL. Doing so is particularly useful when you want to indicate that a given response should be provided for requests that follow a given pattern. Also, you can use regular expressions when you need to use patterns and not exact values both for your test and your server side tests.

The following example shows how to use regular expressions to write a request:

org.springframework.cloud.contract.spec.Contract.make {

request {

method('GET')

url $(consumer(~/\/[0-9]{2}/), producer('/12'))

}

response {

status OK()

body(

id: $(anyNumber()),

surname: $(

consumer('Kowalsky'),

producer(regex('[a-zA-Z]+'))

),

name: 'Jan',

created: $(consumer('2014-02-02 12:23:43'), producer(execute('currentDate(it)'))),

correlationId: value(consumer('5d1f9fef-e0dc-4f3d-a7e4-72d2220dd827'),

producer(regex('[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12}'))

)

)

headers {

header 'Content-Type': 'text/plain'

}

}

}

You can also provide only one side of the communication with a regular expression. If you do so, then the contract engine automatically provides the generated string that matches the provided regular expression. The following code shows an example:

org.springframework.cloud.contract.spec.Contract.make {

request {

method 'PUT'

url value(consumer(regex('/foo/[0-9]{5}')))

body([

requestElement: $(consumer(regex('[0-9]{5}')))

])

headers {

header('header', $(consumer(regex('application\\/vnd\\.fraud\\.v1\\+json;.\*'))))

}

}

response {

status OK()

body([

responseElement: $(producer(regex('[0-9]{7}')))

])

headers {

contentType("application/vnd.fraud.v1+json")

}

}

}

In the preceding example, the opposite side of the communication has the respective data generated for request and response.

Spring Cloud Contract comes with a series of predefined regular expressions that you can use in your contracts, as shown in the following example:

**protected** **static** **final** Pattern TRUE\_OR\_FALSE = Pattern.compile(/(true|false)/)

**protected** **static** **final** Pattern ALPHA\_NUMERIC = Pattern.compile('[a-zA-Z0-9]+')

**protected** **static** **final** Pattern ONLY\_ALPHA\_UNICODE = Pattern.compile(/[\p{L}]\*/)

**protected** **static** **final** Pattern NUMBER = Pattern.compile('-?(\\d\*\\.\\d+|\\d+)')

**protected** **static** **final** Pattern INTEGER = Pattern.compile('-?(\\d+)')

**protected** **static** **final** Pattern POSITIVE\_INT = Pattern.compile('([1-9]\\d\*)')

**protected** **static** **final** Pattern DOUBLE = Pattern.compile('-?(\\d\*\\.\\d+)')

**protected** **static** **final** Pattern HEX = Pattern.compile('[a-fA-F0-9]+')

**protected** **static** **final** Pattern IP\_ADDRESS = Pattern.compile('([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.([01]?\\d\\d?|2[0-4]\\d|25[0-5])')

**protected** **static** **final** Pattern HOSTNAME\_PATTERN = Pattern.compile('((http[s]?|ftp):/)/?([^:/\\s]+)(:[0-9]{1,5})?')

**protected** **static** **final** Pattern EMAIL = Pattern.compile('[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,6}')

**protected** **static** **final** Pattern URL = UrlHelper.URL

**protected** **static** **final** Pattern HTTPS\_URL = UrlHelper.HTTPS\_URL

**protected** **static** **final** Pattern UUID = Pattern.compile('[a-f0-9]{8}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{12}')

**protected** **static** **final** Pattern ANY\_DATE = Pattern.compile('(\\d\\d\\d\\d)-(0[1-9]|1[012])-(0[1-9]|[12][0-9]|3[01])')

**protected** **static** **final** Pattern ANY\_DATE\_TIME = Pattern.compile('([0-9]{4})-(1[0-2]|0[1-9])-(3[01]|0[1-9]|[12][0-9])T(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9])')

**protected** **static** **final** Pattern ANY\_TIME = Pattern.compile('(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9])')

**protected** **static** **final** Pattern NON\_EMPTY = Pattern.compile(/[\S\s]+/)

**protected** **static** **final** Pattern NON\_BLANK = Pattern.compile(/^\s\*\S[\S\s]\*/)

**protected** **static** **final** Pattern ISO8601\_WITH\_OFFSET = Pattern.compile(/([0-9]{4})-(1[0-2]|0[1-9])-(3[01]|0[1-9]|[12][0-9])T(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9])(\.\d{3})?(Z|[+-][01]\d:[0-5]\d)/)

**protected** **static** Pattern anyOf(String... values){

**return** Pattern.compile(values.collect({"^$it\$"}).join("|"))

}

Pattern onlyAlphaUnicode() {

**return** ONLY\_ALPHA\_UNICODE

}

Pattern alphaNumeric() {

**return** ALPHA\_NUMERIC

}

Pattern number() {

**return** NUMBER

}

Pattern positiveInt() {

**return** POSITIVE\_INT

}

Pattern anyBoolean() {

**return** TRUE\_OR\_FALSE

}

Pattern anInteger() {

**return** INTEGER

}

Pattern aDouble() {

**return** DOUBLE

}

Pattern ipAddress() {

**return** IP\_ADDRESS

}

Pattern hostname() {

**return** HOSTNAME\_PATTERN

}

Pattern email() {

**return** EMAIL

}

Pattern url() {

**return** URL

}

Pattern httpsUrl() {

**return** HTTPS\_URL

}

Pattern uuid(){

**return** UUID

}

Pattern isoDate() {

**return** ANY\_DATE

}

Pattern isoDateTime() {

**return** ANY\_DATE\_TIME

}

Pattern isoTime() {

**return** ANY\_TIME

}

Pattern iso8601WithOffset() {

**return** ISO8601\_WITH\_OFFSET

}

Pattern nonEmpty() {

**return** NON\_EMPTY

}

Pattern nonBlank() {

**return** NON\_BLANK

}

In your contract, you can use it as shown in the following example:

Contract dslWithOptionalsInString = Contract.make {

priority 1

request {

method POST()

url '/users/password'

headers {

contentType(applicationJson())

}

body(

email: $(consumer(optional(regex(email()))), producer('abc@abc.com')),

callback\_url: $(consumer(regex(hostname())), producer('http://partners.com'))

)

}

response {

status 404

headers {

contentType(applicationJson())

}

body(

code: value(consumer("123123"), producer(optional("123123"))),

message: "User not found by email = [${value(producer(regex(email())), consumer('not.existing@user.com'))}]"

)

}

}

**95.5.3 Passing Optional Parameters**

|  |  |
| --- | --- |
| [Important] | **Important** |
| This section is valid only for Groovy DSL. Check out the [Section 95.5.7, “Dynamic Properties in the Matchers Sections”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#contract-matchers) section for YAML examples of a similar feature. |

It is possible to provide optional parameters in your contract. However, you can provide optional parameters only for the following:

* *STUB* side of the Request
* *TEST* side of the Response

The following example shows how to provide optional parameters:

org.springframework.cloud.contract.spec.Contract.make {

priority 1

request {

method 'POST'

url '/users/password'

headers {

contentType(applicationJson())

}

body(

email: $(consumer(optional(regex(email()))), producer('abc@abc.com')),

callback\_url: $(consumer(regex(hostname())), producer('http://partners.com'))

)

}

response {

status 404

headers {

header 'Content-Type': 'application/json'

}

body(

code: value(consumer("123123"), producer(optional("123123")))

)

}

}

By wrapping a part of the body with the optional() method, you create a regular expression that must be present 0 or more times.

If you use Spock for, the following test would be generated from the previous example:

"""

given:

def request = given()

.header("Content-Type", "application/json")

.body('''{"email":"abc@abc.com","callback\_url":"http://partners.com"}''')

when:

def response = given().spec(request)

.post("/users/password")

then:

response.statusCode == 404

response.header('Content-Type') == 'application/json'

and:

DocumentContext parsedJson = JsonPath.parse(response.body.asString())

assertThatJson(parsedJson).field("['code']").matches("(123123)?")

"""

The following stub would also be generated:

'''

{

"request" : {

"url" : "/users/password",

"method" : "POST",

"bodyPatterns" : [ {

"matchesJsonPath" : "$[?(@.['email'] =~ /([a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\\\\.[a-zA-Z]{2,6})?/)]"

}, {

"matchesJsonPath" : "$[?(@.['callback\_url'] =~ /((http[s]?|ftp):\\\\/)\\\\/?([^:\\\\/\\\\s]+)(:[0-9]{1,5})?/)]"

} ],

"headers" : {

"Content-Type" : {

"equalTo" : "application/json"

}

}

},

"response" : {

"status" : 404,

"body" : "{\\"code\\":\\"123123\\",\\"message\\":\\"User not found by email == [not.existing*@user.com]\\"}",*

"headers" : {

"Content-Type" : "application/json"

}

},

"priority" : 1

}

'''

**95.5.4 Executing Custom Methods on the Server Side**

|  |  |
| --- | --- |
| [Important] | **Important** |
| This section is valid only for Groovy DSL. Check out the [Section 95.5.7, “Dynamic Properties in the Matchers Sections”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#contract-matchers) section for YAML examples of a similar feature. |

You can define a method call that executes on the server side during the test. Such a method can be added to the class defined as "baseClassForTests" in the configuration. The following code shows an example of the contract portion of the test case:

org.springframework.cloud.contract.spec.Contract.make {

request {

method 'PUT'

url $(consumer(regex('^/api/[0-9]{2}$')), producer('/api/12'))

headers {

header 'Content-Type': 'application/json'

}

body '''\

[{

"text": "Gonna see you at Warsaw"

}]

'''

}

response {

body (

path: $(consumer('/api/12'), producer(regex('^/api/[0-9]{2}$'))),

correlationId: $(consumer('1223456'), producer(execute('isProperCorrelationId($it)')))

)

status OK()

}

}

The following code shows the base class portion of the test case:

**abstract** **class** BaseMockMvcSpec **extends** Specification {

def setup() {

RestAssuredMockMvc.standaloneSetup(**new** PairIdController())

}

**void** isProperCorrelationId(Integer correlationId) {

assert correlationId == 123456

}

**void** isEmpty(String value) {

assert value == null

}

}

|  |  |
| --- | --- |
| [Important] | **Important** |
| You cannot use both a String and execute to perform concatenation. For example, calling header('Authorization', 'Bearer ' + execute('authToken()')) leads to improper results. Instead, call header('Authorization', execute('authToken()')) and ensure that the authToken() method returns everything you need. |

The type of the object read from the JSON can be one of the following, depending on the JSON path:

* String: If you point to a String value in the JSON.
* JSONArray: If you point to a List in the JSON.
* Map: If you point to a Map in the JSON.
* Number: If you point to Integer, Double etc. in the JSON.
* Boolean: If you point to a Boolean in the JSON.

In the request part of the contract, you can specify that the body should be taken from a method.

|  |  |
| --- | --- |
| [Important] | **Important** |
| You must provide both the consumer and the producer side. The execute part is applied for the whole body - not for parts of it. |

The following example shows how to read an object from JSON:

Contract contractDsl = Contract.make {

request {

method 'GET'

url '/something'

body(

$(c("foo"), p(execute("hashCode()")))

)

}

response {

status OK()

}

}

The preceding example results in calling the hashCode() method in the request body. It should resemble the following code:

*// given:*

MockMvcRequestSpecification request = given()

.body(hashCode());

*// when:*

ResponseOptions response = given().spec(request)

.get("/something");

*// then:*

assertThat(response.statusCode()).isEqualTo(200);

**95.5.5 Referencing the Request from the Response**

The best situation is to provide fixed values, but sometimes you need to reference a request in your response.

If you’re writing contracts using Groovy DSL, you can use the fromRequest() method, which lets you reference a bunch of elements from the HTTP request. You can use the following options:

* fromRequest().url(): Returns the request URL and query parameters.
* fromRequest().query(String key): Returns the first query parameter with a given name.
* fromRequest().query(String key, int index): Returns the nth query parameter with a given name.
* fromRequest().path(): Returns the full path.
* fromRequest().path(int index): Returns the nth path element.
* fromRequest().header(String key): Returns the first header with a given name.
* fromRequest().header(String key, int index): Returns the nth header with a given name.
* fromRequest().body(): Returns the full request body.
* fromRequest().body(String jsonPath): Returns the element from the request that matches the JSON Path.

If you’re using the YAML contract definition you have to use the [Handlebars](http://handlebarsjs.com/) {{{ }}} notation with custom, Spring Cloud Contract functions to achieve this.

* {{{ request.url }}}: Returns the request URL and query parameters.
* {{{ request.query.key.[index] }}}: Returns the nth query parameter with a given name. E.g. for key foo, first entry {{{ request.query.foo.[0] }}}
* {{{ request.path }}}: Returns the full path.
* {{{ request.path.[index] }}}: Returns the nth path element. E.g. for first entry `{{{ request.path.[0] }}}
* {{{ request.headers.key }}}: Returns the first header with a given name.
* {{{ request.headers.key.[index] }}}: Returns the nth header with a given name.
* {{{ request.body }}}: Returns the full request body.
* {{{ jsonpath this 'your.json.path' }}}: Returns the element from the request that matches the JSON Path. E.g. for json path $.foo - {{{ jsonpath this '$.foo' }}}

Consider the following contract:

**Groovy DSL.**

Contract contractDsl = Contract.make {

request {

method 'GET'

url('/api/v1/xxxx') {

queryParameters {

parameter("foo", "bar")

parameter("foo", "bar2")

}

}

headers {

header(authorization(), "secret")

header(authorization(), "secret2")

}

body(foo: "bar", baz: 5)

}

response {

status OK()

headers {

header(authorization(), "foo ${fromRequest().header(authorization())} bar")

}

body(

url: fromRequest().url(),

path: fromRequest().path(),

pathIndex: fromRequest().path(1),

param: fromRequest().query("foo"),

paramIndex: fromRequest().query("foo", 1),

authorization: fromRequest().header("Authorization"),

authorization2: fromRequest().header("Authorization", 1),

fullBody: fromRequest().body(),

responseFoo: fromRequest().body('$.foo'),

responseBaz: fromRequest().body('$.baz'),

responseBaz2: "Bla bla ${fromRequest().body('$.foo')} bla bla"

)

}

}

**YAML.**

request:

method: GET

url: /api/v1/xxxx

queryParameters:

foo:

- bar

- bar2

headers:

Authorization:

- secret

- secret2

body:

foo: bar

baz: 5

response:

status: 200

headers:

Authorization: "foo {{{ request.headers.Authorization.0 }}} bar"

body:

url: "{{{ request.url }}}"

path: "{{{ request.path }}}"

pathIndex: "{{{ request.path.1 }}}"

param: "{{{ request.query.foo }}}"

paramIndex: "{{{ request.query.foo.1 }}}"

authorization: "{{{ request.headers.Authorization.0 }}}"

authorization2: "{{{ request.headers.Authorization.1 }}"

fullBody: "{{{ request.body }}}"

responseFoo: "{{{ jsonpath this '$.foo' }}}"

responseBaz: "{{{ jsonpath this '$.baz' }}}"

responseBaz2: "Bla bla {{{ jsonpath this '$.foo' }}} bla bla"

Running a JUnit test generation leads to a test that resembles the following example:

*// given:*

MockMvcRequestSpecification request = given()

.header("Authorization", "secret")

.header("Authorization", "secret2")

.body("{\"foo\":\"bar\",\"baz\":5}");

*// when:*

ResponseOptions response = given().spec(request)

.queryParam("foo","bar")

.queryParam("foo","bar2")

.get("/api/v1/xxxx");

*// then:*

assertThat(response.statusCode()).isEqualTo(200);

assertThat(response.header("Authorization")).isEqualTo("foo secret bar");

*// and:*

DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());

assertThatJson(parsedJson).field("['fullBody']").isEqualTo("{\"foo\":\"bar\",\"baz\":5}");

assertThatJson(parsedJson).field("['authorization']").isEqualTo("secret");

assertThatJson(parsedJson).field("['authorization2']").isEqualTo("secret2");

assertThatJson(parsedJson).field("['path']").isEqualTo("/api/v1/xxxx");

assertThatJson(parsedJson).field("['param']").isEqualTo("bar");

assertThatJson(parsedJson).field("['paramIndex']").isEqualTo("bar2");

assertThatJson(parsedJson).field("['pathIndex']").isEqualTo("v1");

assertThatJson(parsedJson).field("['responseBaz']").isEqualTo(5);

assertThatJson(parsedJson).field("['responseFoo']").isEqualTo("bar");

assertThatJson(parsedJson).field("['url']").isEqualTo("/api/v1/xxxx?foo=bar&foo=bar2");

assertThatJson(parsedJson).field("['responseBaz2']").isEqualTo("Bla bla bar bla bla");

As you can see, elements from the request have been properly referenced in the response.

The generated WireMock stub should resemble the following example:

**{**

"request" : **{**

"urlPath" : "/api/v1/xxxx"**,**

"method" : "POST"**,**

"headers" : **{**

"Authorization" : **{**

"equalTo" : "secret2"

**}**

**},**

"queryParameters" : **{**

"foo" : **{**

"equalTo" : "bar2"

**}**

**},**

"bodyPatterns" : **[** **{**

"matchesJsonPath" : "$[?(@.['baz'] == 5)]"

**},** **{**

"matchesJsonPath" : "$[?(@.['foo'] == 'bar')]"

**}** **]**

**},**

"response" : **{**

"status" : 200**,**

"body" : "{\"authorization\":\"{{{request.headers.Authorization.[0]}}}\",\"path\":\"{{{request.path}}}\",\"responseBaz\":{{{jsonpath this '$.baz'}}} ,\"param\":\"{{{request.query.foo.[0]}}}\",\"pathIndex\":\"{{{request.path.[1]}}}\",\"responseBaz2\":\"Bla bla {{{jsonpath this '$.foo'}}} bla bla\",\"responseFoo\":\"{{{jsonpath this '$.foo'}}}\",\"authorization2\":\"{{{request.headers.Authorization.[1]}}}\",\"fullBody\":\"{{{escapejsonbody}}}\",\"url\":\"{{{request.url}}}\",\"paramIndex\":\"{{{request.query.foo.[1]}}}\"}"**,**

"headers" : **{**

"Authorization" : "{{{request.headers.Authorization.[0]}}};foo"

**},**

"transformers" : **[** "response-template" **]**

**}**

**}**

Sending a request such as the one presented in the request part of the contract results in sending the following response body:

**{**

"url" : "/api/v1/xxxx?foo=bar&foo=bar2"**,**

"path" : "/api/v1/xxxx"**,**

"pathIndex" : "v1"**,**

"param" : "bar"**,**

"paramIndex" : "bar2"**,**

"authorization" : "secret"**,**

"authorization2" : "secret2"**,**

"fullBody" : "{\"foo\":\"bar\",\"baz\":5}"**,**

"responseFoo" : "bar"**,**

"responseBaz" : 5**,**

"responseBaz2" : "Bla bla bar bla bla"

**}**

|  |  |
| --- | --- |
| [Important] | **Important** |
| This feature works only with WireMock having a version greater than or equal to 2.5.1. The Spring Cloud Contract Verifier uses WireMock’sresponse-template response transformer. It uses Handlebars to convert the Mustache {{{ }}} templates into proper values. Additionally, it registers two helper functions: |

* escapejsonbody: Escapes the request body in a format that can be embedded in a JSON.
* jsonpath: For a given parameter, find an object in the request body.

**95.5.6 Registering Your Own WireMock Extension**

WireMock lets you register custom extensions. By default, Spring Cloud Contract registers the transformer, which lets you reference a request from a response. If you want to provide your own extensions, you can register an implementation of theorg.springframework.cloud.contract.verifier.dsl.wiremock.WireMockExtensions interface. Since we use the spring.factories extension approach, you can create an entry in META-INF/spring.factories file similar to the following:

org.springframework.cloud.contract.verifier.dsl.wiremock.WireMockExtensions=\

org.springframework.cloud.contract.stubrunner.provider.wiremock.TestWireMockExtensions

org.springframework.cloud.contract.spec.ContractConverter=\

org.springframework.cloud.contract.stubrunner.TestCustomYamlContractConverter

The following is an example of a custom extension:

**TestWireMockExtensions.groovy.**

**package** org.springframework.cloud.contract.verifier.dsl.wiremock

**import** com.github.tomakehurst.wiremock.extension.Extension

**/\*\***

**\* Extension that registers the default transformer and the custom one**

**\*/**

**class** TestWireMockExtensions **implements** WireMockExtensions {

*@Override*

List<Extension> extensions() {

**return** [

**new** DefaultResponseTransformer(),

**new** CustomExtension()

]

}

}

**class** CustomExtension **implements** Extension {

*@Override*

String getName() {

**return** "foo-transformer"

}

}

|  |  |
| --- | --- |
| [Important] | **Important** |
| Remember to override the applyGlobally() method and set it to false if you want the transformation to be applied only for a mapping that explicitly requires it. |

**95.5.7 Dynamic Properties in the Matchers Sections**

If you work with [Pact](https://docs.pact.io/), the following discussion may seem familiar. Quite a few users are used to having a separation between the body and setting the dynamic parts of a contract.

You can use the bodyMatchers section for two reasons:

* Define the dynamic values that should end up in a stub. You can set it in the request or inputMessage part of your contract.
* Verify the result of your test. This section is present in the response or outputMessage side of the contract.

Currently, Spring Cloud Contract Verifier supports only JSON Path-based matchers with the following matching possibilities:

**Groovy DSL**

* For the stubs(in tests on the Consumer’s side):
  + byEquality(): The value taken from the consumer’s request via the provided JSON Path must be equal to the value provided in the contract.
  + byRegex(…​): The value taken from the consumer’s request via the provided JSON Path must match the regex.
  + byDate(): The value taken from the consumer’s request via the provided JSON Path must match the regex for an ISO Date value.
  + byTimestamp(): The value taken from the consumer’s request via the provided JSON Path must match the regex for an ISO DateTime value.
  + byTime(): The value taken from the consumer’s request via the provided JSON Path must match the regex for an ISO Time value.
* For the verification(in generated tests on the Producer’s side):
  + byEquality(): The value taken from the producer’s response via the provided JSON Path must be equal to the provided value in the contract.
  + byRegex(…​): The value taken from the producer’s response via the provided JSON Path must match the regex.
  + byDate(): The value taken from the producer’s response via the provided JSON Path must match the regex for an ISO Date value.
  + byTimestamp(): The value taken from the producer’s response via the provided JSON Path must match the regex for an ISO DateTime value.
  + byTime(): The value taken from the producer’s response via the provided JSON Path must match the regex for an ISO Time value.
  + byType(): The value taken from the producer’s response via the provided JSON Path needs to be of the same type as the type defined in the body of the response in the contract. byType can take a closure, in which you can set minOccurrence and maxOccurrence. That way, you can assert the size of the flattened collection. To check the size of an unflattened collection, use a custom method with the byCommand(…​) testMatcher.
  + byCommand(…​): The value taken from the producer’s response via the provided JSON Path is passed as an input to the custom method that you provide. For example, byCommand('foo($it)') results in calling a foo method to which the value matching the JSON Path gets passed. The type of the object read from the JSON can be one of the following, depending on the JSON path:
    - String: If you point to a String value.
    - JSONArray: If you point to a List.
    - Map: If you point to a Map.
    - Number: If you point to Integer, Double, or other kind of number.
    - Boolean: If you point to a Boolean.
  + byNull(): The value taken from the response via the provided JSON Path must be null

**YAML.***Please read the Groovy section for detailed explanation of what the types mean*

For YAML the structure of a matcher looks like this

- path: $.foo

type: by\_regex

value: bar

Or if you want to use one of the predefined regular expressions[only\_alpha\_unicode, number, any\_boolean, ip\_address, hostname, email, url, uuid, iso\_date, iso\_date\_time, iso\_time, iso\_8601\_with\_offset, non\_empty, non\_blank]:

- path: $.foo

type: by\_regex

predefined: only\_alpha\_unicode

Below you can find the allowed list of `type`s.

* For stubMatchers:
  + by\_equality
  + by\_regex
  + by\_date
  + by\_timestamp
  + by\_time
* For testMatchers:
  + by\_equality
  + by\_regex
  + by\_date
  + by\_timestamp
  + by\_time
  + by\_type
    - there are 2 additional fields accepted: minOccurrence and maxOccurrence.
  + by\_command
  + by\_null

Consider the following example:

**Groovy DSL.**

Contract contractDsl = Contract.make {

request {

method 'GET'

urlPath '/get'

body([

duck: 123,

alpha: "abc",

number: 123,

aBoolean: true,

date: "2017-01-01",

dateTime: "2017-01-01T01:23:45",

time: "01:02:34",

valueWithoutAMatcher: "foo",

valueWithTypeMatch: "string",

key: [

'complex.key' : 'foo'

]

])

bodyMatchers {

jsonPath('$.duck', byRegex("[0-9]{3}"))

jsonPath('$.duck', byEquality())

jsonPath('$.alpha', byRegex(onlyAlphaUnicode()))

jsonPath('$.alpha', byEquality())

jsonPath('$.number', byRegex(number()))

jsonPath('$.aBoolean', byRegex(anyBoolean()))

jsonPath('$.date', byDate())

jsonPath('$.dateTime', byTimestamp())

jsonPath('$.time', byTime())

jsonPath("\$.['key'].['complex.key']", byEquality())

}

headers {

contentType(applicationJson())

}

}

response {

status OK()

body([

duck: 123,

alpha: "abc",

number: 123,

positiveInteger: 1234567890,

negativeInteger: -1234567890,

positiveDecimalNumber: 123.4567890,

negativeDecimalNumber: -123.4567890,

aBoolean: true,

date: "2017-01-01",

dateTime: "2017-01-01T01:23:45",

time: "01:02:34",

valueWithoutAMatcher: "foo",

valueWithTypeMatch: "string",

valueWithMin: [

1,2,3

],

valueWithMax: [

1,2,3

],

valueWithMinMax: [

1,2,3

],

valueWithMinEmpty: [],

valueWithMaxEmpty: [],

key: [

'complex.key' : 'foo'

],

nullValue: null

])

bodyMatchers {

*// asserts the jsonpath value against manual regex*

jsonPath('$.duck', byRegex("[0-9]{3}"))

*// asserts the jsonpath value against the provided value*

jsonPath('$.duck', byEquality())

*// asserts the jsonpath value against some default regex*

jsonPath('$.alpha', byRegex(onlyAlphaUnicode()))

jsonPath('$.alpha', byEquality())

jsonPath('$.number', byRegex(number()))

jsonPath('$.positiveInteger', byRegex(anInteger()))

jsonPath('$.negativeInteger', byRegex(anInteger()))

jsonPath('$.positiveDecimalNumber', byRegex(aDouble()))

jsonPath('$.negativeDecimalNumber', byRegex(aDouble()))

jsonPath('$.aBoolean', byRegex(anyBoolean()))

*// asserts vs inbuilt time related regex*

jsonPath('$.date', byDate())

jsonPath('$.dateTime', byTimestamp())

jsonPath('$.time', byTime())

*// asserts that the resulting type is the same as in response body*

jsonPath('$.valueWithTypeMatch', byType())

jsonPath('$.valueWithMin', byType {

*// results in verification of size of array (min 1)*

minOccurrence(1)

})

jsonPath('$.valueWithMax', byType {

*// results in verification of size of array (max 3)*

maxOccurrence(3)

})

jsonPath('$.valueWithMinMax', byType {

*// results in verification of size of array (min 1 & max 3)*

minOccurrence(1)

maxOccurrence(3)

})

jsonPath('$.valueWithMinEmpty', byType {

*// results in verification of size of array (min 0)*

minOccurrence(0)

})

jsonPath('$.valueWithMaxEmpty', byType {

*// results in verification of size of array (max 0)*

maxOccurrence(0)

})

*// will execute a method `assertThatValueIsANumber`*

jsonPath('$.duck', byCommand('assertThatValueIsANumber($it)'))

jsonPath("\$.['key'].['complex.key']", byEquality())

jsonPath('$.nullValue', byNull())

}

headers {

contentType(applicationJson())

header('Some-Header', $(c('someValue'), p(regex('[a-zA-Z]{9}'))))

}

}

}

**YAML.**

request:

method: GET

urlPath: /get

body:

duck: 123

alpha: "abc"

number: 123

aBoolean: true

date: "2017-01-01"

dateTime: "2017-01-01T01:23:45"

time: "01:02:34"

valueWithoutAMatcher: "foo"

valueWithTypeMatch: "string"

key:

"complex.key": 'foo'

nullValue: null

matchers:

headers:

- key: Content-Type

regex: "application/json.\*"

body:

- path: $.duck

type: by\_regex

value: "[0-9]{3}"

- path: $.duck

type: by\_equality

- path: $.alpha

type: by\_regex

predefined: only\_alpha\_unicode

- path: $.alpha

type: by\_equality

- path: $.number

type: by\_regex

predefined: number

- path: $.aBoolean

type: by\_regex

predefined: any\_boolean

- path: $.date

type: by\_date

- path: $.dateTime

type: by\_timestamp

- path: $.time

type: by\_time

- path: "$.['key'].['complex.key']"

type: by\_equality

- path: $.nullvalue

type: by\_null

headers:

Content-Type: application/json

response:

status: 200

body:

duck: 123

alpha: "abc"

number: 123

aBoolean: true

date: "2017-01-01"

dateTime: "2017-01-01T01:23:45"

time: "01:02:34"

valueWithoutAMatcher: "foo"

valueWithTypeMatch: "string"

valueWithMin:

- 1

- 2

- 3

valueWithMax:

- 1

- 2

- 3

valueWithMinMax:

- 1

- 2

- 3

valueWithMinEmpty: []

valueWithMaxEmpty: []

key:

'complex.key' : 'foo'

nulValue: null

matchers:

headers:

- key: Content-Type

regex: "application/json.\*"

body:

- path: $.duck

type: by\_regex

value: "[0-9]{3}"

- path: $.duck

type: by\_equality

- path: $.alpha

type: by\_regex

predefined: only\_alpha\_unicode

- path: $.alpha

type: by\_equality

- path: $.number

type: by\_regex

predefined: number

- path: $.aBoolean

type: by\_regex

predefined: any\_boolean

- path: $.date

type: by\_date

- path: $.dateTime

type: by\_timestamp

- path: $.time

type: by\_time

- path: $.valueWithTypeMatch

type: by\_type

- path: $.valueWithMin

type: by\_type

minOccurrence: 1

- path: $.valueWithMax

type: by\_type

maxOccurrence: 3

- path: $.valueWithMinMax

type: by\_type

minOccurrence: 1

maxOccurrence: 3

- path: $.valueWithMinEmpty

type: by\_type

minOccurrence: 0

- path: $.valueWithMaxEmpty

type: by\_type

maxOccurrence: 0

- path: $.duck

type: by\_command

value: assertThatValueIsANumber($it)

- path: $.nullValue

type: by\_null

value: null

headers:

Content-Type: application/json

In the preceding example, you can see the dynamic portions of the contract in the matchers sections. For the request part, you can see that, for all fields butvalueWithoutAMatcher, the values of the regular expressions that the stub should contain are explicitly set. For the valueWithoutAMatcher, the verification takes place in the same way as without the use of matchers. In that case, the test performs an equality check.

For the response side in the bodyMatchers section, we define the dynamic parts in a similar manner. The only difference is that the byType matchers are also present. The verifier engine checks four fields to verify whether the response from the test has a value for which the JSON path matches the given field, is of the same type as the one defined in the response body, and passes the following check (based on the method being called):

* For $.valueWithTypeMatch, the engine checks whether the type is the same.
* For $.valueWithMin, the engine check the type and asserts whether the size is greater than or equal to the minimum occurrence.
* For $.valueWithMax, the engine checks the type and asserts whether the size is smaller than or equal to the maximum occurrence.
* For $.valueWithMinMax, the engine checks the type and asserts whether the size is between the min and maximum occurrence.

The resulting test would resemble the following example (note that an and section separates the autogenerated assertions and the assertion from matchers):

*// given:*

MockMvcRequestSpecification request = given()

.header("Content-Type", "application/json")

.body("{\"duck\":123,\"alpha\":\"abc\",\"number\":123,\"aBoolean\":true,\"date\":\"2017-01-01\",\"dateTime\":\"2017-01-01T01:23:45\",\"time\":\"01:02:34\",\"valueWithoutAMatcher\":\"foo\",\"valueWithTypeMatch\":\"string\",\"key\":{\"complex.key\":\"foo\"}}");

*// when:*

ResponseOptions response = given().spec(request)

.get("/get");

*// then:*

assertThat(response.statusCode()).isEqualTo(200);

assertThat(response.header("Content-Type")).matches("application/json.\*");

*// and:*

DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());

assertThatJson(parsedJson).field("['valueWithoutAMatcher']").isEqualTo("foo");

*// and:*

assertThat(parsedJson.read("$.duck", String.**class**)).matches("[0-9]{3}");

assertThat(parsedJson.read("$.duck", Integer.**class**)).isEqualTo(123);

assertThat(parsedJson.read("$.alpha", String.**class**)).matches("[\\p{L}]\*");

assertThat(parsedJson.read("$.alpha", String.**class**)).isEqualTo("abc");

assertThat(parsedJson.read("$.number", String.**class**)).matches("-?(\\d\*\\.\\d+|\\d+)");

assertThat(parsedJson.read("$.aBoolean", String.**class**)).matches("(true|false)");

assertThat(parsedJson.read("$.date", String.**class**)).matches("(\\d\\d\\d\\d)-(0[1-9]|1[012])-(0[1-9]|[12][0-9]|3[01])");

assertThat(parsedJson.read("$.dateTime", String.**class**)).matches("([0-9]{4})-(1[0-2]|0[1-9])-(3[01]|0[1-9]|[12][0-9])T(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9])");

assertThat(parsedJson.read("$.time", String.**class**)).matches("(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9])");

assertThat((Object) parsedJson.read("$.valueWithTypeMatch")).isInstanceOf(java.lang.String.**class**);

assertThat((Object) parsedJson.read("$.valueWithMin")).isInstanceOf(java.util.List.**class**);

assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMin", java.util.Collection.**class**)).as("$.valueWithMin").hasSizeGreaterThanOrEqualTo(1);

assertThat((Object) parsedJson.read("$.valueWithMax")).isInstanceOf(java.util.List.**class**);

assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMax", java.util.Collection.**class**)).as("$.valueWithMax").hasSizeLessThanOrEqualTo(3);

assertThat((Object) parsedJson.read("$.valueWithMinMax")).isInstanceOf(java.util.List.**class**);

assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMinMax", java.util.Collection.**class**)).as("$.valueWithMinMax").hasSizeBetween(1, 3);

assertThat((Object) parsedJson.read("$.valueWithMinEmpty")).isInstanceOf(java.util.List.**class**);

assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMinEmpty", java.util.Collection.**class**)).as("$.valueWithMinEmpty").hasSizeGreaterThanOrEqualTo(0);

assertThat((Object) parsedJson.read("$.valueWithMaxEmpty")).isInstanceOf(java.util.List.**class**);

assertThat((java.lang.Iterable) parsedJson.read("$.valueWithMaxEmpty", java.util.Collection.**class**)).as("$.valueWithMaxEmpty").hasSizeLessThanOrEqualTo(0);

assertThatValueIsANumber(parsedJson.read("$.duck"));

assertThat(parsedJson.read("$.['key'].['complex.key']", String.**class**)).isEqualTo("foo");

|  |  |
| --- | --- |
| [Important] | **Important** |
| Notice that, for the byCommand method, the example calls the assertThatValueIsANumber. This method must be defined in the test base class or be statically imported to your tests. Notice that the byCommand call was converted to assertThatValueIsANumber(parsedJson.read("$.duck"));. That means that the engine took the method name and passed the proper JSON path as a parameter to it. |

The resulting WireMock stub is in the following example:

'''

**{**

"request" : **{**

"urlPath" : "/get"**,**

"method" : "POST"**,**

"headers" : **{**

"Content-Type" : **{**

"matches" : "application/json.\*"

**}**

**},**

"bodyPatterns" : **[** **{**

"matchesJsonPath" : "$[?(@.['valueWithoutAMatcher'] == 'foo')]"

**},** **{**

"matchesJsonPath" : "$[?(@.['valueWithTypeMatch'] == 'string')]"

**},** **{**

"matchesJsonPath" : "$.['list'].['some'].['nested'][?(@.['anothervalue'] == 4)]"

**},** **{**

"matchesJsonPath" : "$.['list'].['someother'].['nested'][?(@.['anothervalue'] == 4)]"

**},** **{**

"matchesJsonPath" : "$.['list'].['someother'].['nested'][?(@.['json'] == 'with value')]"

**},** **{**

"matchesJsonPath" : "$[?(@.duck =~ /([0-9]{3})/)]"

**},** **{**

"matchesJsonPath" : "$[?(@.duck == 123)]"

**},** **{**

"matchesJsonPath" : "$[?(@.alpha =~ /([\\\\p{L}]\*)/)]"

**},** **{**

"matchesJsonPath" : "$[?(@.alpha == 'abc')]"

**},** **{**

"matchesJsonPath" : "$[?(@.number =~ /(-?(\\\\d\*\\\\.\\\\d+|\\\\d+))/)]"

**},** **{**

"matchesJsonPath" : "$[?(@.aBoolean =~ /((true|false))/)]"

**},** **{**

"matchesJsonPath" : "$[?(@.date =~ /((\\\\d\\\\d\\\\d\\\\d)-(0[1-9]|1[012])-(0[1-9]|[12][0-9]|3[01]))/)]"

**},** **{**

"matchesJsonPath" : "$[?(@.dateTime =~ /(([0-9]{4})-(1[0-2]|0[1-9])-(3[01]|0[1-9]|[12][0-9])T(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9]))/)]"

**},** **{**

"matchesJsonPath" : "$[?(@.time =~ /((2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9]))/)]"

**},** **{**

"matchesJsonPath" : "$.list.some.nested[?(@.json =~ /(.\*)/)]"

**}** **]**

**},**

"response" : **{**

"status" : 200**,**

"body" : "{\\"date\\":\\"2017-01-01\\",\\"dateTime\\":\\"2017-01-01T01:23:45\\",\\"number\\":123,\\"aBoolean\\":true,\\"duck\\":123,\\"alpha\\":\\"abc\\",\\"valueWithMin\\":[1,2,3],\\"time\\":\\"01:02:34\\",\\"valueWithTypeMatch\\":\\"string\\",\\"valueWithMax\\":[1,2,3],\\"valueWithMinMax\\":[1,2,3],\\"valueWithoutAMatcher\\":\\"foo\\"}"**,**

"headers" : **{**

"Content-Type" : "application/json"

**}**

**}**

**}**

'''

|  |  |
| --- | --- |
| [Important] | **Important** |
| If you use a matcher, then the part of the request and response that the matcher addresses with the JSON Path gets removed from the assertion. In the case of verifying a collection, you must create matchers for **all** the elements of the collection. |

Consider the following example:

Contract.make {

request {

method 'GET'

url("/foo")

}

response {

status OK()

body(events: [[

operation : 'EXPORT',

eventId : '16f1ed75-0bcc-4f0d-a04d-3121798faf99',

status : 'OK'

], [

operation : 'INPUT\_PROCESSING',

eventId : '3bb4ac82-6652-462f-b6d1-75e424a0024a',

status : 'OK'

]

]

)

bodyMatchers {

jsonPath('$.events[0].operation', byRegex('.+'))

jsonPath('$.events[0].eventId', byRegex('^([a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12})$'))

jsonPath('$.events[0].status', byRegex('.+'))

}

}

}

The preceding code leads to creating the following test (the code block shows only the assertion section):

and:

DocumentContext parsedJson = JsonPath.parse(response.body.asString())

assertThatJson(parsedJson).array("['events']").contains("['eventId']").isEqualTo("16f1ed75-0bcc-4f0d-a04d-3121798faf99")

assertThatJson(parsedJson).array("['events']").contains("['operation']").isEqualTo("EXPORT")

assertThatJson(parsedJson).array("['events']").contains("['operation']").isEqualTo("INPUT\_PROCESSING")

assertThatJson(parsedJson).array("['events']").contains("['eventId']").isEqualTo("3bb4ac82-6652-462f-b6d1-75e424a0024a")

assertThatJson(parsedJson).array("['events']").contains("['status']").isEqualTo("OK")

and:

assertThat(parsedJson.read("\$.events[0].operation", String.**class**)).matches(".+")

assertThat(parsedJson.read("\$.events[0].eventId", String.**class**)).matches("^([a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12})\$")

assertThat(parsedJson.read("\$.events[0].status", String.**class**)).matches(".+")

As you can see, the assertion is malformed. Only the first element of the array got asserted. In order to fix this, you should apply the assertion to the whole $.eventscollection and assert it with the byCommand(…​) method.

**95.6 JAX-RS Support**

The Spring Cloud Contract Verifier supports the JAX-RS 2 Client API. The base class needs to define protected WebTarget webTarget and server initialization. The only option for testing JAX-RS API is to start a web server. Also, a request with a body needs to have a content type set. Otherwise, the default of application/octet-stream gets used.

In order to use JAX-RS mode, use the following settings:

testMode == 'JAXRSCLIENT'

The following example shows a generated test API:

'''

*// when:*

Response response = webTarget

.path("/users")

.queryParam("limit", "10")

.queryParam("offset", "20")

.queryParam("filter", "email")

.queryParam("sort", "name")

.queryParam("search", "55")

.queryParam("age", "99")

.queryParam("name", "Denis.Stepanov")

.queryParam("email", "bob@email.com")

.request()

.method("GET");

String responseAsString = response.readEntity(String.**class**);

*// then:*

assertThat(response.getStatus()).isEqualTo(200);

*// and:*

DocumentContext parsedJson = JsonPath.parse(responseAsString);

assertThatJson(parsedJson).field("['property1']").isEqualTo("a");

'''

**95.7 Async Support**

If you’re using asynchronous communication on the server side (your controllers are returning Callable, DeferredResult, and so on), then, inside your contract, you must provide an async() method in the response section. The following code shows an example:

**Groovy DSL.**

org.springframework.cloud.contract.spec.Contract.make {

request {

method GET()

url '/get'

}

response {

status OK()

body 'Passed'

async()

}

}

**YAML.**

response:

async: true

**95.8 Working with Context Paths**

Spring Cloud Contract supports context paths.

|  |  |
| --- | --- |
| [Important] | **Important** |
| The only change needed to fully support context paths is the switch on the **PRODUCER** side. Also, the autogenerated tests must use **EXPLICIT** mode. The consumer side remains untouched. In order for the generated test to pass, you must use **EXPLICIT** mode. |

**Maven.**

<plugin>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions>

<configuration>

<testMode>EXPLICIT</testMode>

</configuration>

</plugin>

**Gradle.**

contracts {

testMode = 'EXPLICIT'

}

That way, you generate a test that **DOES NOT** use MockMvc. It means that you generate real requests and you need to setup your generated test’s base class to work on a real socket.

Consider the following contract:

org.springframework.cloud.contract.spec.Contract.make {

request {

method 'GET'

url '/my-context-path/url'

}

response {

status OK()

}

}

The following example shows how to set up a base class and Rest Assured:

**import** io.restassured.RestAssured;

**import** org.junit.Before;

**import** org.springframework.boot.web.server.LocalServerPort;

**import** org.springframework.boot.test.context.SpringBootTest;

*@SpringBootTest(classes = ContextPathTestingBaseClass.class, webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT)*

**class** ContextPathTestingBaseClass {

*@LocalServerPort* **int** port;

*@Before*

**public** **void** setup() {

RestAssured.baseURI = "http://localhost";

RestAssured.port = **this**.port;

}

}

If you do it this way:

* All of your requests in the autogenerated tests are sent to the real endpoint with your context path included (for example, /my-context-path/url).
* Your contracts reflect that you have a context path. Your generated stubs also have that information (for example, in the stubs, you have to call /my-context-path/url).

**95.9 Working with Web Flux**

Spring Cloud Contract requires the usage of EXPLICIT mode in your generated tests to work with Web Flux.

**Maven.**

<plugin>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions>

<configuration>

<testMode>EXPLICIT</testMode>

</configuration>

</plugin>

**Gradle.**

contracts {

testMode = 'EXPLICIT'

}

The following example shows how to set up a base class and Rest Assured for Web Flux:

*@RunWith(SpringRunner.class)*

*@SpringBootTest(classes = BeerRestBase.Config.class,*

*webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT,*

*properties = "server.port=0")*

**public** **abstract** **class** BeerRestBase {

*// your tests go here*

*// in this config class you define all controllers and mocked services*

*@Configuration*

*@EnableAutoConfiguration*

**static** **class** Config {

*@Bean*

PersonCheckingService personCheckingService() {

**return** personToCheck -> personToCheck.age >= 20;

}

*@Bean*

ProducerController producerController() {

**return** **new** ProducerController(personCheckingService());

}

}

}

**95.10 Messaging Top-Level Elements**

The DSL for messaging looks a little bit different than the one that focuses on HTTP. The following sections explain the differences:

* [Section 95.10.1, “Output Triggered by a Method”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#contract-dsl-output-triggered-method)
* [Section 95.10.2, “Output Triggered by a Message”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#contract-dsl-output-triggered-message)
* [Section 95.10.3, “Consumer/Producer”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#contract-dsl-consumer-producer)
* [Section 95.10.4, “Common”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#contract-dsl-common)

**95.10.1 Output Triggered by a Method**

The output message can be triggered by calling a method (such as a Scheduler when a was started and a message was sent), as shown in the following example:

**Groovy DSL.**

def dsl = Contract.make {

*// Human readable description*

description 'Some description'

*// Label by means of which the output message can be triggered*

label 'some\_label'

*// input to the contract*

input {

*// the contract will be triggered by a method*

triggeredBy('bookReturnedTriggered()')

}

*// output message of the contract*

outputMessage {

*// destination to which the output message will be sent*

sentTo('output')

*// the body of the output message*

body('''{ "bookName" : "foo" }''')

*// the headers of the output message*

headers {

header('BOOK-NAME', 'foo')

}

}

}

**YAML.**

# Human readable description

description: Some description

# Label by means of which the output message can be triggered

label: some\_label

input:

# the contract will be triggered by a method

triggeredBy: bookReturnedTriggered()

# output message of the contract

outputMessage:

# destination to which the output message will be sent

sentTo: output

# the body of the output message

body:

bookName: foo

# the headers of the output message

headers:

BOOK-NAME: foo

In the previous example case, the output message is sent to output if a method called bookReturnedTriggered is executed. On the message **publisher’s** side, we generate a test that calls that method to trigger the message. On the **consumer** side, you can use the some\_label to trigger the message.

**95.10.2 Output Triggered by a Message**

The output message can be triggered by receiving a message, as shown in the following example:

**Groovy DSL.**

def dsl = Contract.make {

description 'Some Description'

label 'some\_label'

*// input is a message*

input {

*// the message was received from this destination*

messageFrom('input')

*// has the following body*

messageBody([

bookName: 'foo'

])

*// and the following headers*

messageHeaders {

header('sample', 'header')

}

}

outputMessage {

sentTo('output')

body([

bookName: 'foo'

])

headers {

header('BOOK-NAME', 'foo')

}

}

}

**YAML.**

# Human readable description

description: Some description

# Label by means of which the output message can be triggered

label: some\_label

# input is a message

input:

messageFrom: input

# has the following body

messageBody:

bookName: 'foo'

# and the following headers

messageHeaders:

sample: 'header'

# output message of the contract

outputMessage:

# destination to which the output message will be sent

sentTo: output

# the body of the output message

body:

bookName: foo

# the headers of the output message

headers:

BOOK-NAME: foo

In the preceding example, the output message is sent to output if a proper message is received on the input destination. On the message **publisher’s** side, the engine generates a test that sends the input message to the defined destination. On the **consumer** side, you can either send a message to the input destination or use a label (some\_label in the example) to trigger the message.

**95.10.3 Consumer/Producer**

|  |  |
| --- | --- |
| [Important] | **Important** |
| This section is valid only for Groovy DSL. |

In HTTP, you have a notion of client/stub and `server/test notation. You can also use those paradigms in messaging. In addition, Spring Cloud Contract Verifier also provides the consumer and producer methods, as presented in the following example (note that you can use either $ or value methods to provide consumerand producer parts):

Contract.make {

label 'some\_label'

input {

messageFrom value(consumer('jms:output'), producer('jms:input'))

messageBody([

bookName: 'foo'

])

messageHeaders {

header('sample', 'header')

}

}

outputMessage {

sentTo $(consumer('jms:input'), producer('jms:output'))

body([

bookName: 'foo'

])

}

}

**95.10.4 Common**

In the input or outputMessage section you can call assertThat with the name of a method (e.g. assertThatMessageIsOnTheQueue()) that you have defined in the base class or in a static import. Spring Cloud Contract will execute that method in the generated test.

**95.11 Multiple Contracts in One File**

You can define multiple contracts in one file. Such a contract might resemble the following example:

**Groovy DSL.**

**import** org.springframework.cloud.contract.spec.Contract

[

Contract.make {

name("should post a user")

request {

method 'POST'

url('/users/1')

}

response {

status OK()

}

},

Contract.make {

request {

method 'POST'

url('/users/2')

}

response {

status OK()

}

}

]

**YAML.**

---

name: should post a user

request:

method: POST

url: /users/1

response:

status: 200

---

request:

method: POST

url: /users/2

response:

status: 200

In the preceding example, one contract has the name field and the other does not. This leads to generation of two tests that look more or less like this:

**package** org.springframework.cloud.contract.verifier.tests.com.hello;

**import** com.example.TestBase;

**import** com.jayway.jsonpath.DocumentContext;

**import** com.jayway.jsonpath.JsonPath;

**import** com.jayway.restassured.module.mockmvc.specification.MockMvcRequestSpecification;

**import** com.jayway.restassured.response.ResponseOptions;

**import** org.junit.Test;

**import** **static** com.jayway.restassured.module.mockmvc.RestAssuredMockMvc.\*;

**import** **static** com.toomuchcoding.jsonassert.JsonAssertion.assertThatJson;

**import** **static** org.assertj.core.api.Assertions.assertThat;

**public** **class** V1Test **extends** TestBase {

*@Test*

**public** **void** validate\_should\_post\_a\_user() **throws** Exception {

*// given:*

MockMvcRequestSpecification request = given();

*// when:*

ResponseOptions response = given().spec(request)

.post("/users/1");

*// then:*

assertThat(response.statusCode()).isEqualTo(200);

}

*@Test*

**public** **void** validate\_withList\_1() **throws** Exception {

*// given:*

MockMvcRequestSpecification request = given();

*// when:*

ResponseOptions response = given().spec(request)

.post("/users/2");

*// then:*

assertThat(response.statusCode()).isEqualTo(200);

}

}

Notice that, for the contract that has the name field, the generated test method is named validate\_should\_post\_a\_user. For the one that does not have the name, it is called validate\_withList\_1. It corresponds to the name of the file WithList.groovy and the index of the contract in the list.

The generated stubs is shown in the following example:

should post a user.json

1\_WithList.json

As you can see, the first file got the name parameter from the contract. The second got the name of the contract file (WithList.groovy) prefixed with the index (in this case, the contract had an index of 1 in the list of contracts in the file).

|  |
| --- |
| [Tip] |
| As you can see, it is much better if you name your contracts because doing so makes your tests far more meaningful. |

**95.12 Generating Spring REST Docs snippets from the contracts**

When you want to include the requests and responses of your API using Spring REST Docs, you only need to make some minor changes to your setup if you are using MockMvc and RestAssuredMockMvc. Simply include the following dependencies if you haven’t already.

**Maven.**

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-contract-verifier</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework.restdocs</groupId>

<artifactId>spring-restdocs-mockmvc</artifactId>

<optional>true</optional>

</dependency>

**Gradle.**

testCompile 'org.springframework.cloud:spring-cloud-starter-contract-verifier'

testCompile 'org.springframework.restdocs:spring-restdocs-mockmvc'

Next you need to make some changes to your base class like the following example.

**package** com.example.fraud;

**import** io.restassured.module.mockmvc.RestAssuredMockMvc;

**import** org.junit.Before;

**import** org.junit.Rule;

**import** org.junit.rules.TestName;

**import** org.junit.runner.RunWith;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.boot.test.context.SpringBootTest;

**import** org.springframework.restdocs.JUnitRestDocumentation;

**import** org.springframework.test.context.junit4.SpringRunner;

**import** org.springframework.test.web.servlet.setup.MockMvcBuilders;

**import** org.springframework.web.context.WebApplicationContext;

**import** **static** org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.document;

**import** **static** org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.documentationConfiguration;

*@RunWith(SpringRunner.class)*

*@SpringBootTest(classes = Application.class)*

**public** **abstract** **class** FraudBaseWithWebAppSetup {

**private** **static** **final** String OUTPUT = "target/generated-snippets";

*@Rule*

**public** JUnitRestDocumentation restDocumentation = **new** JUnitRestDocumentation(OUTPUT);

*@Rule* **public** TestName testName = **new** TestName();

*@Autowired*

**private** WebApplicationContext context;

*@Before*

**public** **void** setup() {

RestAssuredMockMvc.mockMvc(MockMvcBuilders.webAppContextSetup(**this**.context)

.apply(documentationConfiguration(**this**.restDocumentation))

.alwaysDo(document(getClass().getSimpleName() + "\_" + testName.getMethodName()))

.build());

}

**protected** **void** assertThatRejectionReasonIsNull(Object rejectionReason) {

assert rejectionReason == null;

}

}

In case you are using the standalone setup, you can set up RestAssuredMockMvc like this:

**package** com.example.fraud;

**import** io.restassured.module.mockmvc.RestAssuredMockMvc;

**import** org.junit.Before;

**import** org.junit.Rule;

**import** org.junit.rules.TestName;

**import** org.springframework.restdocs.JUnitRestDocumentation;

**import** org.springframework.test.web.servlet.setup.MockMvcBuilders;

**import** **static** org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.document;

**import** **static** org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.documentationConfiguration;

**public** **abstract** **class** FraudBaseWithStandaloneSetup {

**private** **static** **final** String OUTPUT = "target/generated-snippets";

*@Rule*

**public** JUnitRestDocumentation restDocumentation = **new** JUnitRestDocumentation(OUTPUT);

*@Rule* **public** TestName testName = **new** TestName();

*@Before*

**public** **void** setup() {

RestAssuredMockMvc.standaloneSetup(MockMvcBuilders.standaloneSetup(**new** FraudDetectionController())

.apply(documentationConfiguration(**this**.restDocumentation))

.alwaysDo(document(getClass().getSimpleName() + "\_" + testName.getMethodName())));

}

}

|  |
| --- |
| [Tip] |
| You don’t need to specify the output directory for the generated snippets since version 1.2.0.RELEASE of Spring REST Docs. |

**96. Customization**

|  |  |
| --- | --- |
| [Important] | **Important** |
| This section is valid only for Groovy DSL |

You can customize the Spring Cloud Contract Verifier by extending the DSL, as shown in the remainder of this section.

**96.1 Extending the DSL**

You can provide your own functions to the DSL. The key requirement for this feature is to maintain the static compatibility. Later in this document, you can see examples of:

* Creating a JAR with reusable classes.
* Referencing of these classes in the DSLs.

You can find the full example [here](https://github.com/spring-cloud-samples/spring-cloud-contract-samples).

**96.1.1 Common JAR**

The following examples show three classes that can be reused in the DSLs.

**PatternUtils** contains functions used by both the **consumer** and the **producer**.

**package** com.example;

**import** java.util.regex.Pattern;

**/\*\***

**\* If you want to use {@link Pattern} directly in your tests**

**\* then you can create a class resembling this one. It can**

**\* contain all the {@link Pattern} you want to use in the DSL.**

**\***

**\* <pre>**

**\* {@code**

**\* request {**

**\* body(**

**\* [ age: $(c(PatternUtils.oldEnough()))]**

**\* )**

**\* }**

**\* </pre>**

**\***

**\* Notice that we're using both {@code $()} for dynamic values**

**\* and {@code c()} for the consumer side.**

**\***

**\* @author Marcin Grzejszczak**

**\*/**

*//tag::impl[]*

**public** **class** PatternUtils {

**public** **static** String tooYoung() {

*//remove::start[]*

**return** "[0-1][0-9]";

*//remove::end[return]*

}

**public** **static** Pattern oldEnough() {

*//remove::start[]*

**return** Pattern.compile("[2-9][0-9]");

*//remove::end[return]*

}

**/\*\***

**\* Makes little sense but it's just an example ;)**

**\*/**

**public** **static** Pattern ok() {

*//remove::start[]*

**return** Pattern.compile("OK");

*//remove::end[return]*

}

}

*//end::impl[]*

**ConsumerUtils** contains functions used by the **consumer**.

**package** com.example;

**import** org.springframework.cloud.contract.spec.internal.ClientDslProperty;

**/\*\***

**\* DSL Properties passed to the DSL from the consumer's perspective.**

**\* That means that on the input side {@code Request} for HTTP**

**\* or {@code Input} for messaging you can have a regular expression.**

**\* On the {@code Response} for HTTP or {@code Output} for messaging**

**\* you have to have a concrete value.**

**\***

**\* @author Marcin Grzejszczak**

**\*/**

*//tag::impl[]*

**public** **class** ConsumerUtils {

**/\*\***

**\* Consumer side property. By using the {@link ClientDslProperty}**

**\* you can omit most of boilerplate code from the perspective**

**\* of dynamic values. Example**

**\***

**\* <pre>**

**\* {@code**

**\* request {**

**\* body(**

**\* [ age: $(ConsumerUtils.oldEnough())]**

**\* )**

**\* }**

**\* </pre>**

**\***

**\* That way it's in the implementation that we decide what value we will pass to the consumer**

**\* and which one to the producer.**

**\***

**\* @author Marcin Grzejszczak**

**\*/**

**public** **static** ClientDslProperty oldEnough() {

*//remove::start[]*

*// this example is not the best one and*

*// theoretically you could just pass the regex instead of `ServerDslProperty` but*

*// it's just to show some new tricks :)*

**return** **new** ClientDslProperty(PatternUtils.oldEnough(), 40);

*//remove::end[return]*

}

}

*//end::impl[]*

**ProducerUtils** contains functions used by the **producer**.

**package** com.example;

**import** org.springframework.cloud.contract.spec.internal.ServerDslProperty;

**/\*\***

**\* DSL Properties passed to the DSL from the producer's perspective.**

**\* That means that on the input side {@code Request} for HTTP**

**\* or {@code Input} for messaging you have to have a concrete value.**

**\* On the {@code Response} for HTTP or {@code Output} for messaging**

**\* you can have a regular expression.**

**\***

**\* @author Marcin Grzejszczak**

**\*/**

*//tag::impl[]*

**public** **class** ProducerUtils {

**/\*\***

**\* Producer side property. By using the {@link ProducerUtils}**

**\* you can omit most of boilerplate code from the perspective**

**\* of dynamic values. Example**

**\***

**\* <pre>**

**\* {@code**

**\* response {**

**\* body(**

**\* [ status: $(ProducerUtils.ok())]**

**\* )**

**\* }**

**\* </pre>**

**\***

**\* That way it's in the implementation that we decide what value we will pass to the consumer**

**\* and which one to the producer.**

**\*/**

**public** **static** ServerDslProperty ok() {

*// this example is not the best one and*

*// theoretically you could just pass the regex instead of `ServerDslProperty` but*

*// it's just to show some new tricks :)*

**return** **new** ServerDslProperty( PatternUtils.ok(), "OK");

}

}

*//end::impl[]*

**96.1.2 Adding the Dependency to the Project**

In order for the plugins and IDE to be able to reference the common JAR classes, you need to pass the dependency to your project.

**96.1.3 Test the Dependency in the Project’s Dependencies**

First, add the common jar dependency as a test dependency. Because your contracts files are available on the test resources path, the common jar classes automatically become visible in your Groovy files. The following examples show how to test the dependency:

**Maven.**

<dependency>

<groupId>com.example</groupId>

<artifactId>beer-common</artifactId>

<version>${project.version}</version>

<scope>test</scope>

</dependency>

**Gradle.**

testCompile("com.example:beer-common:0.0.1-SNAPSHOT")

**96.1.4 Test a Dependency in the Plugin’s Dependencies**

Now, you must add the dependency for the plugin to reuse at runtime, as shown in the following example:

**Maven.**

<plugin>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions>

<configuration>

<packageWithBaseClasses>com.example</packageWithBaseClasses>

<baseClassMappings>

<baseClassMapping>

<contractPackageRegex>.\*intoxication.\*</contractPackageRegex>

<baseClassFQN>com.example.intoxication.BeerIntoxicationBase</baseClassFQN>

</baseClassMapping>

</baseClassMappings>

</configuration>

<dependencies>

<dependency>

<groupId>com.example</groupId>

<artifactId>beer-common</artifactId>

<version>${project.version}</version>

<scope>compile</scope>

</dependency>

</dependencies>

</plugin>

**Gradle.**

classpath "com.example:beer-common:0.0.1-SNAPSHOT"

**96.1.5 Referencing classes in DSLs**

You can now reference your classes in your DSL, as shown in the following example:

**package** contracts.beer.rest

**import** com.example.ConsumerUtils

**import** com.example.ProducerUtils

**import** org.springframework.cloud.contract.spec.Contract

Contract.make {

description("""

Represents a successful scenario of getting a beer

```

given:

client is old enough

when:

he applies **for** a beer

then:

we'll grant him the beer

```

""")

request {

method 'POST'

url '/check'

body(

age: $(ConsumerUtils.oldEnough())

)

headers {

contentType(applicationJson())

}

}

response {

status 200

body("""

{

"status": "${value(ProducerUtils.ok())}"

}

""")

headers {

contentType(applicationJson())

}

}

}

**97. Using the Pluggable Architecture**

You may encounter cases where you have your contracts have been defined in other formats, such as YAML, RAML or PACT. In those cases, you still want to benefit from the automatic generation of tests and stubs. You can add your own implementation for generating both tests and stubs. Also, you can customize the way tests are generated (for example, you can generate tests for other languages) and the way stubs are generated (for example, you can generate stubs for other HTTP server implementations).

**97.1 Custom Contract Converter**

The ContractConverter interface lets you register your own implementation of a contract structure converter. The following code listing shows the ContractConverter interface:

**package** org.springframework.cloud.contract.spec

**/\*\***

**\* Converter to be used to convert FROM {@link File} TO {@link Contract}**

**\* and from {@link Contract} to {@code T}**

**\***

**\* @param <T> - type to which we want to convert the contract**

**\***

**\* @author Marcin Grzejszczak**

**\* @since 1.1.0**

**\*/**

**interface** ContractConverter<T> {

**/\*\***

**\* Should this file be accepted by the converter. Can use the file extension**

**\* to check if the conversion is possible.**

**\***

**\* @param file - file to be considered for conversion**

**\* @return - {@code true} if the given implementation can convert the file**

**\*/**

**boolean** isAccepted(File file)

**/\*\***

**\* Converts the given {@link File} to its {@link Contract} representation**

**\***

**\* @param file - file to convert**

**\* @return - {@link Contract} representation of the file**

**\*/**

Collection<Contract> convertFrom(File file)

**/\*\***

**\* Converts the given {@link Contract} to a {@link T} representation**

**\***

**\* @param contract - the parsed contract**

**\* @return - {@link T} the type to which we do the conversion**

**\*/**

T convertTo(Collection<Contract> contract)

}

Your implementation must define the condition on which it should start the conversion. Also, you must define how to perform that conversion in both directions.

|  |  |
| --- | --- |
| [Important] | **Important** |
| Once you create your implementation, you must create a /META-INF/spring.factories file in which you provide the fully qualified name of your implementation. |

The following example shows a typical spring.factories file:

org.springframework.cloud.contract.spec.ContractConverter=\

org.springframework.cloud.contract.verifier.converter.YamlContractConverter

**97.1.1 Pact Converter**

Spring Cloud Contract includes support for [Pact](https://docs.pact.io/) representation of contracts up until v4. Instead of using the Groovy DSL, you can use Pact files. In this section, we present how to add Pact support for your project. Note however that not all functionality is supported. Starting with v3 you can combine multiple matcher for the same element; you can use matchers for the body, headers, request and path; and you can use value generators. Spring Cloud Contract currently only supports multiple matchers that are combined using the AND rule logic. Next to that the request and path matchers are skipped during the conversion. When using a date, time or datetime value generator with a given format, the given format will be skipped and the ISO format will be used.

**97.1.2 Pact Contract**

Consider following example of a Pact contract, which is a file under the src/test/resources/contracts folder.

{

"provider": {

"name": "Provider"

},

"consumer": {

"name": "Consumer"

},

"interactions": [

{

"description": "",

"request": {

"method": "PUT",

"path": "/fraudcheck",

"headers": {

"Content-Type": "application/vnd.fraud.v1+json"

},

"body": {

"clientId": "1234567890",

"loanAmount": 99999

},

"generators": {

"body": {

"$.clientId": {

"type": "Regex",

"regex": "[0-9]{10}"

}

}

},

"matchingRules": {

"header": {

"Content-Type": {

"matchers": [

{

"match": "regex",

"regex": "application/vnd\\.fraud\\.v1\\+json.\*"

}

],

"combine": "AND"

}

},

"body" : {

"$.clientId": {

"matchers": [

{

"match": "regex",

"regex": "[0-9]{10}"

}

],

"combine": "AND"

}

}

}

},

"response": {

"status": 200,

"headers": {

"Content-Type": "application/vnd.fraud.v1+json;charset=UTF-8"

},

"body": {

"fraudCheckStatus": "FRAUD",

"rejectionReason": "Amount too high"

},

"matchingRules": {

"header": {

"Content-Type": {

"matchers": [

{

"match": "regex",

"regex": "application/vnd\\.fraud\\.v1\\+json.\*"

}

],

"combine": "AND"

}

},

"body": {

"$.fraudCheckStatus": {

"matchers": [

{

"match": "regex",

"regex": "FRAUD"

}

],

"combine": "AND"

}

}

}

}

}

],

"metadata": {

"pact-specification": {

"version": "3.0.0"

},

"pact-jvm": {

"version": "3.5.13"

}

}

}

The remainder of this section about using Pact refers to the preceding file.

**97.1.3 Pact for Producers**

On the producer side, you must add two additional dependencies to your plugin configuration. One is the Spring Cloud Contract Pact support, and the other represents the current Pact version that you use.

**Maven.**

<plugin>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-contract-maven-plugin</artifactId>

<version>${spring-cloud-contract.version}</version>

<extensions>true</extensions>

<configuration>

<packageWithBaseClasses>com.example.fraud</packageWithBaseClasses>

</configuration>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-contract-pact</artifactId>

<version>${spring-cloud-contract.version}</version>

</dependency>

</dependencies>

</plugin>

**Gradle.**

classpath "org.springframework.cloud:spring-cloud-contract-pact:${findProperty('verifierVersion') ?: verifierVersion}"

When you execute the build of your application, a test will be generated. The generated test might be as follows:

*@Test*

**public** **void** validate\_shouldMarkClientAsFraud() **throws** Exception {

*// given:*

MockMvcRequestSpecification request = given()

.header("Content-Type", "application/vnd.fraud.v1+json")

.body("{\"clientId\":\"1234567890\",\"loanAmount\":99999}");

*// when:*

ResponseOptions response = given().spec(request)

.put("/fraudcheck");

*// then:*

assertThat(response.statusCode()).isEqualTo(200);

assertThat(response.header("Content-Type")).matches("application/vnd\\.fraud\\.v1\\+json.\*");

*// and:*

DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());

assertThatJson(parsedJson).field("['rejectionReason']").isEqualTo("Amount too high");

*// and:*

assertThat(parsedJson.read("$.fraudCheckStatus", String.**class**)).matches("FRAUD");

}

The corresponding generated stub might be as follows:

{

"id" : "996ae5ae-6834-4db6-8fac-358ca187ab62",

"uuid" : "996ae5ae-6834-4db6-8fac-358ca187ab62",

"request" : {

"url" : "/fraudcheck",

"method" : "PUT",

"headers" : {

"Content-Type" : {

"matches" : "application/vnd\\.fraud\\.v1\\+json.\*"

}

},

"bodyPatterns" : [ {

"matchesJsonPath" : "$[?(@.['loanAmount'] == 99999)]"

}, {

"matchesJsonPath" : "$[?(@.clientId =~ /([0-9]{10})/)]"

} ]

},

"response" : {

"status" : 200,

"body" : "{\"fraudCheckStatus\":\"FRAUD\",\"rejectionReason\":\"Amount too high\"}",

"headers" : {

"Content-Type" : "application/vnd.fraud.v1+json;charset=UTF-8"

},

"transformers" : [ "response-template" ]

},

}

**97.1.4 Pact for Consumers**

On the producer side, you must add two additional dependencies to your project dependencies. One is the Spring Cloud Contract Pact support, and the other represents the current Pact version that you use.

**Maven.**

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-contract-pact</artifactId>

<scope>test</scope>

</dependency>

**Gradle.**

testCompile "org.springframework.cloud:spring-cloud-contract-pact"

**97.2 Using the Custom Test Generator**

If you want to generate tests for languages other than Java or you are not happy with the way the verifier builds Java tests, you can register your own implementation.

The SingleTestGenerator interface lets you register your own implementation. The following code listing shows the SingleTestGenerator interface:

**package** org.springframework.cloud.contract.verifier.builder

**import** org.springframework.cloud.contract.verifier.config.ContractVerifierConfigProperties

**import** org.springframework.cloud.contract.verifier.file.ContractMetadata

**/\*\***

**\* Builds a single test.**

**\***

**\* @since 1.1.0**

**\*/**

**interface** SingleTestGenerator {

**/\*\***

**\* Creates contents of a single test class in which all test scenarios from**

**\* the contract metadata should be placed.**

**\***

**\* @param properties - properties passed to the plugin**

**\* @param listOfFiles - list of parsed contracts with additional metadata**

**\* @param className - the name of the generated test class**

**\* @param classPackage - the name of the package in which the test class should be stored**

**\* @param includedDirectoryRelativePath - relative path to the included directory**

**\* @return contents of a single test class**

**\*/**

String buildClass(ContractVerifierConfigProperties properties, Collection<ContractMetadata> listOfFiles,

String className, String classPackage, String includedDirectoryRelativePath)

**/\*\***

**\* Extension that should be appended to the generated test class. E.g. {@code .java} or {@code .php}**

**\***

**\* @param properties - properties passed to the plugin**

**\*/**

String fileExtension(ContractVerifierConfigProperties properties)

}

Again, you must provide a spring.factories file, such as the one shown in the following example:

org.springframework.cloud.contract.verifier.builder.SingleTestGenerator=/

com.example.MyGenerator

**97.3 Using the Custom Stub Generator**

If you want to generate stubs for stub servers other than WireMock, you can plug in your own implementation of the StubGenerator interface. The following code listing shows the StubGenerator interface:

**package** org.springframework.cloud.contract.verifier.converter

**import** groovy.transform.CompileStatic

**import** org.springframework.cloud.contract.spec.Contract

**import** org.springframework.cloud.contract.verifier.file.ContractMetadata

**/\*\***

**\* Converts contracts into their stub representation.**

**\***

**\* @since 1.1.0**

**\*/**

*@CompileStatic*

**interface** StubGenerator {

**/\*\***

**\* Returns {@code true} if the converter can handle the file to convert it into a stub.**

**\*/**

**boolean** canHandleFileName(String fileName)

**/\*\***

**\* Returns the collection of converted contracts into stubs. One contract can**

**\* result in multiple stubs.**

**\*/**

Map<Contract, String> convertContents(String rootName, ContractMetadata content)

**/\*\***

**\* Returns the name of the converted stub file. If you have multiple contracts**

**\* in a single file then a prefix will be added to the generated file. If you**

**\* provide the {@link Contract#name} field then that field will override the**

**\* generated file name.**

**\***

**\* Example: name of file with 2 contracts is {@code foo.groovy}, it will be**

**\* converted by the implementation to {@code foo.json}. The recursive file**

**\* converter will create two files {@code 0\_foo.json} and {@code 1\_foo.json}**

**\*/**

String generateOutputFileNameForInput(String inputFileName)

}

Again, you must provide a spring.factories file, such as the one shown in the following example:

# Stub converters

org.springframework.cloud.contract.verifier.converter.StubGenerator=\

org.springframework.cloud.contract.verifier.wiremock.DslToWireMockClientConverter

The default implementation is the WireMock stub generation.

|  |
| --- |
| [Tip] |
| You can provide multiple stub generator implementations. For example, from a single DSL, you can produce both WireMock stubs and Pact files. |

**97.4 Using the Custom Stub Runner**

If you decide to use a custom stub generation, you also need a custom way of running stubs with your different stub provider.

Assume that you use [Moco](https://github.com/dreamhead/moco) to build your stubs and that you have written a stub generator and placed your stubs in a JAR file.

In order for Stub Runner to know how to run your stubs, you have to define a custom HTTP Stub server implementation, which might resemble the following example:

**package** org.springframework.cloud.contract.stubrunner.provider.moco

**import** com.github.dreamhead.moco.bootstrap.arg.HttpArgs

**import** com.github.dreamhead.moco.runner.JsonRunner

**import** com.github.dreamhead.moco.runner.RunnerSetting

**import** groovy.util.logging.Slf4j

**import** org.springframework.cloud.contract.stubrunner.HttpServerStub

**import** org.springframework.util.SocketUtils

*@Slf4j*

**class** MocoHttpServerStub **implements** HttpServerStub {

**private** **boolean** started

**private** JsonRunner runner

**private** **int** port

*@Override*

**int** port() {

**if** (!isRunning()) {

**return** -1

}

**return** port

}

*@Override*

**boolean** isRunning() {

**return** started

}

*@Override*

HttpServerStub start() {

**return** start(SocketUtils.findAvailableTcpPort())

}

*@Override*

HttpServerStub start(**int** port) {

**this**.port = port

**return** **this**

}

*@Override*

HttpServerStub stop() {

**if** (!isRunning()) {

**return** **this**

}

**this**.runner.stop()

**return** **this**

}

*@Override*

HttpServerStub registerMappings(Collection<File> stubFiles) {

List<RunnerSetting> settings = stubFiles.findAll { it.name.endsWith("json") }

.collect {

log.info("Trying to parse [{}]", it.name)

**try** {

**return** RunnerSetting.aRunnerSetting().withStream(it.newInputStream()).build()

} **catch** (Exception e) {

log.warn("Exception occurred while trying to parse file [{}]", it.name, e)

**return** null

}

}.findAll { it }

**this**.runner = JsonRunner.newJsonRunnerWithSetting(settings,

HttpArgs.httpArgs().withPort(**this**.port).build())

**this**.runner.run()

**this**.started = true

**return** **this**

}

*@Override*

String registeredMappings() {

**return** ""

}

*@Override*

**boolean** isAccepted(File file) {

**return** file.name.endsWith(".json")

}

}

Then, you can register it in your spring.factories file, as shown in the following example:

org.springframework.cloud.contract.stubrunner.HttpServerStub=\

org.springframework.cloud.contract.stubrunner.provider.moco.MocoHttpServerStub

Now you can run stubs with Moco.

|  |  |
| --- | --- |
| [Important] | **Important** |
| If you do not provide any implementation, then the default (WireMock) implementation is used. If you provide more than one, the first one on the list is used. |

**97.5 Using the Custom Stub Downloader**

You can customize the way your stubs are downloaded by creating an implementation of the StubDownloaderBuilder interface, as shown in the following example:

**package** com.example;

**class** CustomStubDownloaderBuilder **implements** StubDownloaderBuilder {

*@Override*

**public** StubDownloader build(**final** StubRunnerOptions stubRunnerOptions) {

**return** **new** StubDownloader() {

*@Override*

**public** Map.Entry<StubConfiguration, File> downloadAndUnpackStubJar(

StubConfiguration config) {

File unpackedStubs = retrieveStubs();

**return** **new** AbstractMap.SimpleEntry<>(

**new** StubConfiguration(config.getGroupId(), config.getArtifactId(), version,

config.getClassifier()), unpackedStubs);

}

File retrieveStubs() {

*// here goes your custom logic to provide a folder where all the stubs reside*

}

}

Then you can register it in your spring.factories file, as shown in the following example:

# Example of a custom Stub Downloader Provider

org.springframework.cloud.contract.stubrunner.StubDownloaderBuilder=\

com.example.CustomStubDownloaderBuilder

Now you can pick a folder with the source of your stubs.

|  |  |
| --- | --- |
| [Important] | **Important** |
| If you do not provide any implementation, then the default is used (scan classpath). If you provide the stubsMode = StubRunnerProperties.StubsMode.LOCAL or , stubsMode = StubRunnerProperties.StubsMode.REMOTE then the Aether implementation will be used If you provide more than one, then the first one on the list is used. |

**97.6 Using the SCM Stub Downloader**

Whenever the repositoryRoot starts with a SCM protocol (currently we support only git://), the stub downloader will try to clone the repository and use it as a source of contracts to generate tests or stubs.

Either via environment variables, system properties, properties set inside the plugin or contracts repository configuration you can tweak the downloader’s behaviour. Below you can find the list of properties

**Table 97.1. SCM Stub Downloader properties**

|  |  |  |
| --- | --- | --- |
| Type of a property | Name of the property | Description |
| \* git.branch (plugin prop)  \* stubrunner.properties.git.branch (system prop)  \* STUBRUNNER\_PROPERTIES\_GIT\_BRANCH (env prop) | master | Which branch to checkout |
| \* git.username (plugin prop)  \* stubrunner.properties.git.username (system prop)  \* STUBRUNNER\_PROPERTIES\_GIT\_USERNAME (env prop) |  | Git clone username |
| \* git.password (plugin prop)  \* stubrunner.properties.git.password (system prop)  \* STUBRUNNER\_PROPERTIES\_GIT\_PASSWORD (env prop) |  | Git clone password |
| \* git.no-of-attempts (plugin prop)  \* stubrunner.properties.git.no-of-attempts (system prop)  \* STUBRUNNER\_PROPERTIES\_GIT\_NO\_OF\_ATTEMPTS (env prop) | 10 | Number of attempts to push the commits to origin |
| \* git.wait-between-attempts (Plugin prop)  \* stubrunner.properties.git.wait-between-attempts(system prop)  \* STUBRUNNER\_PROPERTIES\_GIT\_WAIT\_BETWEEN\_ATTEMPTS (env prop) | 1000 | Number of millis to wait between attempts to push the commits to origin |

**97.7 Using the Pact Stub Downloader**

Whenever the repositoryRoot starts with a Pact protocol (starts with pact://), the stub downloader will try to fetch the Pact contract definitions from the Pact Broker. Whatever is set after pact:// will be parsed as the Pact Broker URL.

Either via environment variables, system properties, properties set inside the plugin or contracts repository configuration you can tweak the downloader’s behaviour. Below you can find the list of properties

**Table 97.2. SCM Stub Downloader properties**

|  |  |  |
| --- | --- | --- |
| Name of a property | Default | Description |
| \* pactbroker.host (plugin prop)  \* stubrunner.properties.pactbroker.host (system prop)  \* STUBRUNNER\_PROPERTIES\_PACTBROKER\_HOST (env prop) | Host from URL passed to repositoryRoot | What is the URL of Pact Broker |
| \* pactbroker.port (plugin prop)  \* stubrunner.properties.pactbroker.port (system prop)  \* STUBRUNNER\_PROPERTIES\_PACTBROKER\_PORT (env prop) | Port from URL passed to repositoryRoot | What is the port of Pact Broker |
| \* pactbroker.protocol (plugin prop)  \* stubrunner.properties.pactbroker.protocol (system prop)  \* STUBRUNNER\_PROPERTIES\_PACTBROKER\_PROTOCOL (env prop) | Protocol from URL passed to repositoryRoot | What is the protocol of Pact Broker |
| \* pactbroker.tags (plugin prop)  \* stubrunner.properties.pactbroker.tags (system prop)  \* STUBRUNNER\_PROPERTIES\_PACTBROKER\_TAGS (env prop) | Version of the stub, or latest if version is + | What tags should be used to fetch the stub |
| \* pactbroker.auth.scheme (plugin prop)  \* stubrunner.properties.pactbroker.auth.scheme (system prop)  \* STUBRUNNER\_PROPERTIES\_PACTBROKER\_AUTH\_SCHEME (env prop) | Basic | What kind of authentication should be used to connect to the Pact Broker |
| \* pactbroker.auth.username (plugin prop)  \* stubrunner.properties.pactbroker.auth.username (system prop)  \* STUBRUNNER\_PROPERTIES\_PACTBROKER\_AUTH\_USERNAME (env prop) | The username passed to contractsRepositoryUsername(maven) or contractRepository.username(gradle) | Username used to connect to the Pact Broker |
| \* pactbroker.auth.password (plugin prop)  \* stubrunner.properties.pactbroker.auth.password (system prop)  \* STUBRUNNER\_PROPERTIES\_PACTBROKER\_AUTH\_PASSWORD (env prop) | The password passed to contractsRepositoryPassword(maven) or contractRepository.password(gradle) | Password used to connect to the Pact Broker |
| \* pactbroker.provider-name-with-group-id (plugin prop)  \* stubrunner.properties.pactbroker.provider-name-with-group-id(system prop)  \* STUBRUNNER\_PROPERTIES\_PACTBROKER\_PROVIDER\_NAME\_WITH\_GROUP\_ID(env prop) | false | When true, the provider name will be a combination of groupId:artifactId. If false, just artifactId is used |

**98. Spring Cloud Contract WireMock**

The Spring Cloud Contract WireMock modules let you use [WireMock](http://wiremock.org/) in a Spring Boot application. Check out the [samples](https://github.com/spring-cloud/spring-cloud-contract/tree/master/samples) for more details.

If you have a Spring Boot application that uses Tomcat as an embedded server (which is the default with spring-boot-starter-web), you can addspring-cloud-starter-contract-stub-runner to your classpath and add @AutoConfigureWireMock in order to be able to use Wiremock in your tests. Wiremock runs as a stub server and you can register stub behavior using a Java API or via static JSON declarations as part of your test. The following code shows an example:

*@RunWith(SpringRunner.class)*

*@SpringBootTest(webEnvironment = WebEnvironment.RANDOM\_PORT)*

*@AutoConfigureWireMock(port = 0)*

**public** **class** WiremockForDocsTests {

*// A service that calls out over HTTP*

*@Autowired* **private** Service service;

*// Using the WireMock APIs in the normal way:*

*@Test*

**public** **void** contextLoads() **throws** Exception {

*// Stubbing WireMock*

stubFor(get(urlEqualTo("/resource"))

.willReturn(aResponse().withHeader("Content-Type", "text/plain").withBody("Hello World!")));

*// We're asserting if WireMock responded properly*

assertThat(**this**.service.go()).isEqualTo("Hello World!");

}

}

To start the stub server on a different port use (for example), @AutoConfigureWireMock(port=9999). For a random port, use a value of 0. The stub server port can be bound in the test application context with the "wiremock.server.port" property. Using @AutoConfigureWireMock adds a bean of type WiremockConfiguration to your test application context, where it will be cached in between methods and classes having the same context, the same as for Spring integration tests.

**98.1 Registering Stubs Automatically**

If you use @AutoConfigureWireMock, it registers WireMock JSON stubs from the file system or classpath (by default, from file:src/test/resources/mappings). You can customize the locations using the stubs attribute in the annotation, which can be an Ant-style resource pattern or a directory. In the case of a directory, **\*/**.json is appended. The following code shows an example:

@RunWith(SpringRunner.class)

@SpringBootTest

@AutoConfigureWireMock(stubs="classpath:/stubs")

public class WiremockImportApplicationTests {

@Autowired

private Service service;

@Test

public void contextLoads() throws Exception {

assertThat(this.service.go()).isEqualTo("Hello World!");

}

}

|  |
| --- |
| [Note] |
| Actually, WireMock always loads mappings from src/test/resources/mappings **as well as** the custom locations in the stubs attribute. To change this behavior, you can also specify a files root as described in the next section of this document. |

**98.2 Using Files to Specify the Stub Bodies**

WireMock can read response bodies from files on the classpath or the file system. In that case, you can see in the JSON DSL that the response has a bodyFileNameinstead of a (literal) body. The files are resolved relative to a root directory (by default, src/test/resources/\_\_files). To customize this location you can set the files attribute in the @AutoConfigureWireMock annotation to the location of the parent directory (in other words, \_\_files is a subdirectory). You can use Spring resource notation to refer to file:…​ or classpath:…​ locations. Generic URLs are not supported. A list of values can be given, in which case WireMock resolves the first file that exists when it needs to find a response body.

|  |
| --- |
| [Note] |
| When you configure the files root, it also affects the automatic loading of stubs, because they come from the root location in a subdirectory called "mappings". The value of files has no effect on the stubs loaded explicitly from the stubs attribute. |

**98.3 Alternative: Using JUnit Rules**

For a more conventional WireMock experience, you can use JUnit @Rules to start and stop the server. To do so, use the WireMockSpring convenience class to obtain an Options instance, as shown in the following example:

*@RunWith(SpringRunner.class)*

*@SpringBootTest(webEnvironment = WebEnvironment.RANDOM\_PORT)*

**public** **class** WiremockForDocsClassRuleTests {

*// Start WireMock on some dynamic port*

*// for some reason `dynamicPort()` is not working properly*

*@ClassRule*

**public** **static** WireMockClassRule wiremock = **new** WireMockClassRule(

WireMockSpring.options().dynamicPort());

*// A service that calls out over HTTP to localhost:${wiremock.port}*

*@Autowired*

**private** Service service;

*// Using the WireMock APIs in the normal way:*

*@Test*

**public** **void** contextLoads() **throws** Exception {

*// Stubbing WireMock*

wiremock.stubFor(get(urlEqualTo("/resource"))

.willReturn(aResponse().withHeader("Content-Type", "text/plain").withBody("Hello World!")));

*// We're asserting if WireMock responded properly*

assertThat(**this**.service.go()).isEqualTo("Hello World!");

}

}

The @ClassRule means that the server shuts down after all the methods in this class have been run.

**98.4 Relaxed SSL Validation for Rest Template**

WireMock lets you stub a "secure" server with an "https" URL protocol. If your application wants to contact that stub server in an integration test, it will find that the SSL certificates are not valid (the usual problem with self-installed certificates). The best option is often to re-configure the client to use "http". If that’s not an option, you can ask Spring to configure an HTTP client that ignores SSL validation errors (do so only for tests, of course).

To make this work with minimum fuss, you need to be using the Spring Boot RestTemplateBuilder in your app, as shown in the following example:

*@Bean*

**public** RestTemplate restTemplate(RestTemplateBuilder builder) {

**return** builder.build();

}

You need RestTemplateBuilder because the builder is passed through callbacks to initialize it, so the SSL validation can be set up in the client at that point. This happens automatically in your test if you are using the @AutoConfigureWireMock annotation or the stub runner. If you use the JUnit @Rule approach, you need to add the @AutoConfigureHttpClient annotation as well, as shown in the following example:

*@RunWith(SpringRunner.class)*

*@SpringBootTest("app.baseUrl=https://localhost:6443")*

*@AutoConfigureHttpClient*

**public** **class** WiremockHttpsServerApplicationTests {

*@ClassRule*

**public** **static** WireMockClassRule wiremock = **new** WireMockClassRule(

WireMockSpring.options().httpsPort(6443));

...

}

If you are using spring-boot-starter-test, you have the Apache HTTP client on the classpath and it is selected by the RestTemplateBuilder and configured to ignore SSL errors. If you use the default java.net client, you do not need the annotation (but it won’t do any harm). There is no support currently for other clients, but it may be added in future releases.

To disable the custom RestTemplateBuilder, set the wiremock.rest-template-ssl-enabled property to false.

**98.5 WireMock and Spring MVC Mocks**

Spring Cloud Contract provides a convenience class that can load JSON WireMock stubs into a Spring MockRestServiceServer. The following code shows an example:

*@RunWith(SpringRunner.class)*

*@SpringBootTest(webEnvironment = WebEnvironment.NONE)*

**public** **class** WiremockForDocsMockServerApplicationTests {

*@Autowired*

**private** RestTemplate restTemplate;

*@Autowired*

**private** Service service;

*@Test*

**public** **void** contextLoads() **throws** Exception {

*// will read stubs classpath*

MockRestServiceServer server = WireMockRestServiceServer.with(**this**.restTemplate)

.baseUrl("http://example.org").stubs("classpath:/stubs/resource.json")

.build();

*// We're asserting if WireMock responded properly*

assertThat(**this**.service.go()).isEqualTo("Hello World");

server.verify();

}

}

The baseUrl value is prepended to all mock calls, and the stubs() method takes a stub path resource pattern as an argument. In the preceding example, the stub defined at /stubs/resource.json is loaded into the mock server. If the RestTemplate is asked to visit <http://example.org/>, it gets the responses as being declared at that URL. More than one stub pattern can be specified, and each one can be a directory (for a recursive list of all ".json"), a fixed filename (as in the example above), or an Ant-style pattern. The JSON format is the normal WireMock format, which you can read about in the [WireMock website](http://wiremock.org/docs/stubbing/).

Currently, the Spring Cloud Contract Verifier supports Tomcat, Jetty, and Undertow as Spring Boot embedded servers, and Wiremock itself has "native" support for a particular version of Jetty (currently 9.2). To use the native Jetty, you need to add the native Wiremock dependencies and exclude the Spring Boot container (if there is one).

**98.6 Customization of WireMock configuration**

You can register a bean of org.springframework.cloud.contract.wiremock.WireMockConfigurationCustomizer type in order to customize the WireMock configuration (e.g. add custom transformers). Example:

*@Bean* WireMockConfigurationCustomizer optionsCustomizer() {

**return** **new** WireMockConfigurationCustomizer() {

*@Override* **public** **void** customize(WireMockConfiguration options) {

*// perform your customization here*

}

};

}

**98.7 Generating Stubs using REST Docs**

[Spring REST Docs](https://projects.spring.io/spring-restdocs) can be used to generate documentation (for example in Asciidoctor format) for an HTTP API with Spring MockMvc or WebTestClient or Rest Assured. At the same time that you generate documentation for your API, you can also generate WireMock stubs by using Spring Cloud Contract WireMock. To do so, write your normal REST Docs test cases and use @AutoConfigureRestDocs to have stubs be automatically generated in the REST Docs output directory. The following code shows an example using MockMvc:

*@RunWith(SpringRunner.class)*

*@SpringBootTest*

*@AutoConfigureRestDocs(outputDir = "target/snippets")*

*@AutoConfigureMockMvc*

**public** **class** ApplicationTests {

*@Autowired*

**private** MockMvc mockMvc;

*@Test*

**public** **void** contextLoads() **throws** Exception {

mockMvc.perform(get("/resource"))

.andExpect(content().string("Hello World"))

.andDo(document("resource"));

}

}

This test generates a WireMock stub at "target/snippets/stubs/resource.json". It matches all GET requests to the "/resource" path. The same example with WebTestClient (used for testing Spring WebFlux applications) would look like this:

*@RunWith(SpringRunner.class)*

*@SpringBootTest*

*@AutoConfigureRestDocs(outputDir = "target/snippets")*

*@AutoConfigureWebTestClient*

**public** **class** ApplicationTests {

*@Autowired*

**private** WebTestClient client;

*@Test*

**public** **void** contextLoads() **throws** Exception {

client.get().uri("/resource").exchange()

.expectBody(String.**class**).isEqualTo("Hello World")

.consumeWith(document("resource"));

}

}

Without any additional configuration, these tests create a stub with a request matcher for the HTTP method and all headers except "host" and "content-length". To match the request more precisely (for example, to match the body of a POST or PUT), we need to explicitly create a request matcher. Doing so has two effects:

* Creating a stub that matches only in the way you specify.
* Asserting that the request in the test case also matches the same conditions.

The main entry point for this feature is WireMockRestDocs.verify(), which can be used as a substitute for the document() convenience method, as shown in the following example:

**import** **static** org.springframework.cloud.contract.wiremock.restdocs.WireMockRestDocs.verify;

@RunWith(SpringRunner.class)

@SpringBootTest

@AutoConfigureRestDocs(outputDir = "target/snippets")

@AutoConfigureMockMvc

public class ApplicationTests {

@Autowired

private MockMvc mockMvc;

@Test

public void contextLoads() throws Exception {

mockMvc.perform(post("/resource")

.content("{\"id\":\"123456\",\"message\":\"Hello World\"}"))

.andExpect(status().isOk())

.andDo(verify().jsonPath("$.id")

.stub("resource"));

}

}

This contract specifies that any valid POST with an "id" field receives the response defined in this test. You can chain together calls to .jsonPath() to add additional matchers. If JSON Path is unfamiliar, The [JayWay documentation](https://github.com/jayway/JsonPath) can help you get up to speed. The WebTestClient version of this test has a similar verify() static helper that you insert in the same place.

Instead of the jsonPath and contentType convenience methods, you can also use the WireMock APIs to verify that the request matches the created stub, as shown in the following example:

*@Test*

**public** **void** contextLoads() **throws** Exception {

mockMvc.perform(post("/resource")

.content("{\"id\":\"123456\",\"message\":\"Hello World\"}"))

.andExpect(status().isOk())

.andDo(verify()

.wiremock(WireMock.post(

urlPathEquals("/resource"))

.withRequestBody(matchingJsonPath("$.id"))

.stub("post-resource"));

}

The WireMock API is rich. You can match headers, query parameters, and request body by regex as well as by JSON path. These features can be used to create stubs with a wider range of parameters. The above example generates a stub resembling the following example:

**post-resource.json.**

**{**

"request" : **{**

"url" : "/resource"**,**

"method" : "POST"**,**

"bodyPatterns" : **[** **{**

"matchesJsonPath" : "$.id"

**}]**

**},**

"response" : **{**

"status" : 200**,**

"body" : "Hello World"**,**

"headers" : **{**

"X-Application-Context" : "application:-1"**,**

"Content-Type" : "text/plain"

**}**

**}**

**}**

|  |
| --- |
| [Note] |
| You can use either the wiremock() method or the jsonPath() and contentType() methods to create request matchers, but you can’t use both approaches. |

On the consumer side, you can make the resource.json generated earlier in this section available on the classpath (by <<publishing-stubs-as-jars], for example). After that, you can create a stub using WireMock in a number of different ways, including by using @AutoConfigureWireMock(stubs="classpath:resource.json"), as described earlier in this document.

**98.8 Generating Contracts by Using REST Docs**

You can also generate Spring Cloud Contract DSL files and documentation with Spring REST Docs. If you do so in combination with Spring Cloud WireMock, you get both the contracts and the stubs.

Why would you want to use this feature? Some people in the community asked questions about a situation in which they would like to move to DSL-based contract definition, but they already have a lot of Spring MVC tests. Using this feature lets you generate the contract files that you can later modify and move to folders (defined in your configuration) so that the plugin finds them.

|  |
| --- |
| [Tip] |
| You might wonder why this functionality is in the WireMock module. The functionality is there because it makes sense to generate both the contracts and the stubs. |

Consider the following test:

**this**.mockMvc.perform(post("/foo")

.accept(MediaType.APPLICATION\_PDF)

.accept(MediaType.APPLICATION\_JSON)

.contentType(MediaType.APPLICATION\_JSON)

.content("{\"foo\": 23, \"bar\" : \"baz\" }"))

.andExpect(status().isOk())

.andExpect(content().string("bar"))

*// first WireMock*

.andDo(WireMockRestDocs.verify()

.jsonPath("$[?(@.foo >= 20)]")

.jsonPath("$[?(@.bar in ['baz','bazz','bazzz'])]")

.contentType(MediaType.valueOf("application/json"))

.stub("shouldGrantABeerIfOldEnough"))

*// then Contract DSL documentation*

.andDo(document("index", SpringCloudContractRestDocs.dslContract()));

The preceding test creates the stub presented in the previous section, generating both the contract and a documentation file.

The contract is called index.groovy and might look like the following example:

**import** org.springframework.cloud.contract.spec.Contract

Contract.make {

request {

method 'POST'

url '/foo'

body('''

{"foo": 23 }

''')

headers {

header('''Accept''', '''application/json''')

header('''Content-Type''', '''application/json''')

}

}

response {

status OK()

body('''

bar

''')

headers {

header('''Content-Type''', '''application/json;charset=UTF-8''')

header('''Content-Length''', '''3''')

}

testMatchers {

jsonPath('$[?(@.foo >= 20)]', byType())

}

}

}

The generated document (formatted in Asciidoc in this case) contains a formatted contract. The location of this file would be index/dsl-contract.adoc.

**99. Migrations**

|  |
| --- |
| [Tip] |
| For up to date migration guides please visit the project’s [wiki page](https://github.com/spring-cloud/spring-cloud-contract/wiki/). |

This section covers migrating from one version of Spring Cloud Contract Verifier to the next version. It covers the following versions upgrade paths:

**99.1 1.0.x → 1.1.x**

This section covers upgrading from version 1.0 to version 1.1.

**99.1.1 New structure of generated stubs**

In 1.1.x we have introduced a change to the structure of generated stubs. If you have been using the @AutoConfigureWireMock notation to use the stubs from the classpath, it no longer works. The following example shows how the @AutoConfigureWireMock notation used to work:

@AutoConfigureWireMock(stubs = "classpath:/customer-stubs/mappings", port = 8084)

You must either change the location of the stubs to: classpath:…​/META-INF/groupId/artifactId/version/mappings or use the new classpath-based @AutoConfigureStubRunner, as shown in the following example:

@AutoConfigureWireMock(stubs = "classpath:customer-stubs/META-INF/travel.components/customer-contract/1.0.2-SNAPSHOT/mappings/", port = 8084)

If you do not want to use @AutoConfigureStubRunner and you want to remain with the old structure, set your plugin tasks accordingly. The following example would work for the structure presented in the previous snippet.

**Maven.**

*<!-- start of pom.xml -->*

<properties>

*<!-- we don't want the verifier to do a jar for us -->*

<spring.cloud.contract.verifier.skip>true</spring.cloud.contract.verifier.skip>

</properties>

*<!-- ... -->*

*<!-- You need to set up the assembly plugin -->*

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-assembly-plugin</artifactId>

<executions>

<execution>

<id>stub</id>

<phase>prepare-package</phase>

<goals>

<goal>single</goal>

</goals>

<inherited>false</inherited>

<configuration>

<attach>true</attach>

<descriptor>$../../../../src/assembly/stub.xml</descriptor>

</configuration>

</execution>

</executions>

</plugin>

</plugins>

</build>

*<!-- end of pom.xml -->*

*<!-- start of stub.xml-->*

<assembly

xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3 http://maven.apache.org/xsd/assembly-1.1.3.xsd">

<id>stubs</id>

<formats>

<format>jar</format>

</formats>

<includeBaseDirectory>false</includeBaseDirectory>

<fileSets>

<fileSet>

<directory>${project.build.directory}/snippets/stubs</directory>

<outputDirectory>customer-stubs/mappings</outputDirectory>

<includes>

<include>\*\*/\*</include>

</includes>

</fileSet>

<fileSet>

<directory>$../../../../src/test/resources/contracts</directory>

<outputDirectory>customer-stubs/contracts</outputDirectory>

<includes>

<include>\*\*/\*.groovy</include>

</includes>

</fileSet>

</fileSets>

</assembly>

*<!-- end of stub.xml-->*

**Gradle.**

task copyStubs(type: Copy, dependsOn: 'generateWireMockClientStubs') {

*// Preserve directory structure from 1.0.X of spring-cloud-contract*

from "${project.buildDir}/resources/main/customer-stubs/META-INF/${project.group}/${project.name}/${project.version}"

into "${project.buildDir}/resources/main/customer-stubs"

}

**99.2 1.1.x → 1.2.x**

This section covers upgrading from version 1.1 to version 1.2.

**99.2.1 Custom HttpServerStub**

HttpServerStub includes a method that was not in version 1.1. The method is String registeredMappings() If you have classes that implement HttpServerStub, you now have to implement the registeredMappings() method. It should return a String representing all mappings available in a single HttpServerStub.

See [issue 355](https://github.com/spring-cloud/spring-cloud-contract/issues/355) for more detail.

**99.2.2 New packages for generated tests**

The flow for setting the generated tests package name will look like this:

* Set basePackageForTests
* If basePackageForTests was not set, pick the package from baseClassForTests
* If baseClassForTests was not set, pick packageWithBaseClasses
* If nothing got set, pick the default value: org.springframework.cloud.contract.verifier.tests

See [issue 260](https://github.com/spring-cloud/spring-cloud-contract/issues/260) for more detail.

**99.2.3 New Methods in TemplateProcessor**

In order to add support for fromRequest.path, the following methods had to be added to the TemplateProcessor interface:

* path()
* path(int index)

See [issue 388](https://github.com/spring-cloud/spring-cloud-contract/issues/388) for more detail.

**99.2.4 RestAssured 3.0**

Rest Assured, used in the generated test classes, got bumped to 3.0. If you manually set versions of Spring Cloud Contract and the release train you might see the following exception:

Failed to execute goal org.apache.maven.plugins:maven-compiler-plugin:3.1:testCompile (default-testCompile) on project some-project: Compilation failure: Compilation failure:

[ERROR] /some/path/SomeClass.java:[4,39] package com.jayway.restassured.response does not exist

This exception will occur due to the fact that the tests got generated with an old version of plugin and at test execution time you have an incompatible version of the release train (and vice versa).

Done via [issue 267](https://github.com/spring-cloud/spring-cloud-contract/issues/267)

**99.3 1.2.x → 2.0.x**

**99.3.1 No Camel support**

We will add back Apache Camel support only after this [issue](https://issues.apache.org/jira/browse/CAMEL-11430) gets fixed

**100. Links**

The following links may be helpful when working with Spring Cloud Contract:

* [Spring Cloud Contract Github Repository](https://github.com/spring-cloud/spring-cloud-contract/)
* [Spring Cloud Contract Samples](https://github.com/spring-cloud-samples/spring-cloud-contract-samples/)
* [Spring Cloud Contract Gitter](https://gitter.im/spring-cloud/spring-cloud-contract)
* [Spring Cloud Contract WJUG Presentation by Marcin Grzejszczak](https://www.youtube.com/watch?v=sAAklvxmPmk)

**Part XV. Spring Cloud Vault**

© 2016-2018 The original authors.

|  |
| --- |
| [Note] |
| *Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.* |

Spring Cloud Vault Config provides client-side support for externalized configuration in a distributed system. With [HashiCorp’s Vault](https://www.vaultproject.io/) you have a central place to manage external secret properties for applications across all environments. Vault can manage static and dynamic secrets such as username/password for remote applications/resources and provide credentials for external services such as MySQL, PostgreSQL, Apache Cassandra, MongoDB, Consul, AWS and more.

**101. Quick Start**

**Prerequisites**

To get started with Vault and this guide you need a \*NIX-like operating systems that provides:

* wget, openssl and unzip
* at least Java 7 and a properly configured JAVA\_HOME environment variable

**Install Vault**

$ src/**test**/bash/install\_vault.sh

**Create SSL certificates for Vault**

$ src/**test**/bash/create\_certificates.sh

|  |
| --- |
| [Note] |
| create\_certificates.sh creates certificates in work/ca and a JKS truststore work/keystore.jks. If you want to run Spring Cloud Vault using this quickstart guide you need to configure the truststore the spring.cloud.vault.ssl.trust-store property to file:work/keystore.jks. |

**Start Vault server**

$ src/**test**/bash/local\_run\_vault.sh

Vault is started listening on 0.0.0.0:8200 using the inmem storage and https. Vault is sealed and not initialized when starting up.

|  |
| --- |
| [Note] |
| If you want to run tests, leave Vault uninitialized. The tests will initialize Vault and create a root token 00000000-0000-0000-0000-000000000000. |

If you want to use Vault for your application or give it a try then you need to initialize it first.

$ **export** VAULT\_ADDR="https://localhost:8200"

$ **export** VAULT\_SKIP\_VERIFY=true *# Don't do this for production*

$ vault init

You should see something like:

Key 1: 7149c6a2e16b8833f6eb1e76df03e47f6113a3288b3093faf5033d44f0e70fe701

Key 2: 901c534c7988c18c20435a85213c683bdcf0efcd82e38e2893779f152978c18c02

Key 3: 03ff3948575b1165a20c20ee7c3e6edf04f4cdbe0e82dbff5be49c63f98bc03a03

Key 4: 216ae5cc3ddaf93ceb8e1d15bb9fc3176653f5b738f5f3d1ee00cd7dccbe926e04

Key 5: b2898fc8130929d569c1677ee69dc5f3be57d7c4b494a6062693ce0b1c4d93d805

Initial Root Token: 19aefa97-cccc-bbbb-aaaa-225940e63d76

Vault initialized with 5 keys and a key threshold of 3. Please

securely distribute the above keys. When the Vault is re-sealed,

restarted, or stopped, you must provide at least 3 of these keys

to unseal it again.

Vault does not store the master key. Without at least 3 keys,

your Vault will remain permanently sealed.

Vault will initialize and return a set of unsealing keys and the root token. Pick 3 keys and unseal Vault. Store the Vault token in the VAULT\_TOKEN environment variable.

$ vault unseal (Key 1)

$ vault unseal (Key 2)

$ vault unseal (Key 3)

$ **export** VAULT\_TOKEN=(Root token)

*# Required to run Spring Cloud Vault tests after manual initialization*

$ vault token-create -id="00000000-0000-0000-0000-000000000000" -policy="root"

Spring Cloud Vault accesses different resources. By default, the secret backend is enabled which accesses secret config settings via JSON endpoints.

The HTTP service has resources in the form:

/secret/{application}/{profile}

/secret/{application}

/secret/{defaultContext}/{profile}

/secret/{defaultContext}

where the "application" is injected as the spring.application.name in the SpringApplication (i.e. what is normally "application" in a regular Spring Boot app), "profile" is an active profile (or comma-separated list of properties). Properties retrieved from Vault will be used "as-is" without further prefixing of the property names.

**102. Client Side Usage**

To use these features in an application, just build it as a Spring Boot application that depends on spring-cloud-vault-config (e.g. see the test cases). Example Maven configuration:

**Example 102.1. pom.xml**

<parent>

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

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

<version>2.0.0.RELEASE</version>

<relativePath /> *<!-- lookup parent from repository -->*

</parent>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-vault-config</artifactId>

<version>Finchley.SR1</version>

</dependency>

<dependency>

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

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

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

*<!-- repositories also needed for snapshots and milestones -->*

Then you can create a standard Spring Boot application, like this simple HTTP server:

*@SpringBootApplication*

*@RestController*

**public** **class** Application {

*@RequestMapping("/")*

**public** String home() {

**return** "Hello World!";

}

**public** **static** **void** main(String[] args) {

SpringApplication.run(Application.**class**, args);

}

}

When it runs it will pick up the external configuration from the default local Vault server on port 8200 if it is running. To modify the startup behavior you can change the location of the Vault server using bootstrap.properties (like application.properties but for the bootstrap phase of an application context), e.g.

**Example 102.2. bootstrap.yml**

spring.cloud.vault:

host: localhost

port: 8200

scheme: https

uri: https://localhost:8200

connection-timeout: 5000

read-timeout: 15000

config:

order: -10

* host sets the hostname of the Vault host. The host name will be used for SSL certificate validation
* port sets the Vault port
* scheme setting the scheme to http will use plain HTTP. Supported schemes are http and https.
* uri configure the Vault endpoint with an URI. Takes precedence over host/port/scheme configuration
* connection-timeout sets the connection timeout in milliseconds
* read-timeout sets the read timeout in milliseconds
* config.order sets the order for the property source

Enabling further integrations requires additional dependencies and configuration. Depending on how you have set up Vault you might need additional configuration like[SSL](https://cloud.spring.io/spring-cloud-vault/spring-cloud-vault.html#vault.config.ssl) and [authentication](https://cloud.spring.io/spring-cloud-vault/spring-cloud-vault.html#vault.config.authentication).

If the application imports the spring-boot-starter-actuator project, the status of the vault server will be available via the /health endpoint.

The vault health indicator can be enabled or disabled through the property health.vault.enabled (default true).

**102.1 Authentication**

Vault requires an [authentication mechanism](https://www.vaultproject.io/docs/concepts/auth.html) to [authorize client requests](https://www.vaultproject.io/docs/concepts/tokens.html).

Spring Cloud Vault supports multiple [authentication mechanisms](https://cloud.spring.io/spring-cloud-vault/spring-cloud-vault.html#vault.config.authentication) to authenticate applications with Vault.

For a quickstart, use the root token printed by the [Vault initialization](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#quickstart.vault.start).

**Example 102.3. bootstrap.yml**

spring.cloud.vault:

token: 19aefa97-cccc-bbbb-aaaa-225940e63d76

|  |
| --- |
| [Warning] |
| Consider carefully your security requirements. Static token authentication is fine if you want quickly get started with Vault, but a static token is not protected any further. Any disclosure to unintended parties allows Vault use with the associated token roles. |

**103. Authentication methods**

Different organizations have different requirements for security and authentication. Vault reflects that need by shipping multiple authentication methods. Spring Cloud Vault supports token and AppId authentication.

**103.1 Token authentication**

Tokens are the core method for authentication within Vault. Token authentication requires a static token to be provided using the [Bootstrap Application Context](https://github.com/spring-cloud/spring-cloud-commons/blob/master/docs/src/main/asciidoc/spring-cloud-commons.adoc#the-bootstrap-application-context).

|  |
| --- |
| [Note] |
| Token authentication is the default authentication method. If a token is disclosed an unintended party gains access to Vault and can access secrets for the intended client. |

**Example 103.1. bootstrap.yml**

spring.cloud.vault:

authentication: TOKEN

token: 00000000-0000-0000-0000-000000000000

* authentication setting this value to TOKEN selects the Token authentication method
* token sets the static token to use

See also: [Vault Documentation: Tokens](https://www.vaultproject.io/docs/concepts/tokens.html)

**103.2 AppId authentication**

Vault supports [AppId](https://www.vaultproject.io/docs/auth/app-id.html) authentication that consists of two hard to guess tokens. The AppId defaults to spring.application.name that is statically configured. The second token is the UserId which is a part determined by the application, usually related to the runtime environment. IP address, Mac address or a Docker container name are good examples. Spring Cloud Vault Config supports IP address, Mac address and static UserId’s (e.g. supplied via System properties). The IP and Mac address are represented as Hex-encoded SHA256 hash.

IP address-based UserId’s use the local host’s IP address.

**Example 103.2. bootstrap.yml using SHA256 IP-Address UserId’s**

spring.cloud.vault:

authentication: APPID

app-id:

user-id: IP\_ADDRESS

* authentication setting this value to APPID selects the AppId authentication method
* app-id-path sets the path of the AppId mount to use
* user-id sets the UserId method. Possible values are IP\_ADDRESS, MAC\_ADDRESS or a class name implementing a custom AppIdUserIdMechanism

The corresponding command to generate the IP address UserId from a command line is:

$ echo -n 192.168.99.1 | sha256sum

|  |
| --- |
| [Note] |
| Including the line break of echo leads to a different hash value so make sure to include the -n flag. |

Mac address-based UserId’s obtain their network device from the localhost-bound device. The configuration also allows specifying a network-interface hint to pick the right device. The value of network-interface is optional and can be either an interface name or interface index (0-based).

**Example 103.3. bootstrap.yml using SHA256 Mac-Address UserId’s**

spring.cloud.vault:

authentication: APPID

app-id:

user-id: MAC\_ADDRESS

network-interface: eth0

* network-interface sets network interface to obtain the physical address

The corresponding command to generate the IP address UserId from a command line is:

$ echo -n 0AFEDE1234AC | sha256sum

|  |
| --- |
| [Note] |
| The Mac address is specified uppercase and without colons. Including the line break of echo leads to a different hash value so make sure to include the -n flag. |

**103.2.1 Custom UserId**

The UserId generation is an open mechanism. You can set spring.cloud.vault.app-id.user-id to any string and the configured value will be used as static UserId.

A more advanced approach lets you set spring.cloud.vault.app-id.user-id to a classname. This class must be on your classpath and must implement the org.springframework.cloud.vault.AppIdUserIdMechanism interface and the createUserId method. Spring Cloud Vault will obtain the UserId by calling createUserId each time it authenticates using AppId to obtain a token.

**Example 103.4. bootstrap.yml**

spring.cloud.vault:

authentication: APPID

app-id:

user-id: com.examlple.MyUserIdMechanism

**Example 103.5. MyUserIdMechanism.java**

public class MyUserIdMechanism implements AppIdUserIdMechanism **{**

*@Override*

public String createUserId() **{**

String userId = ...

return userId;

**}**

**}**

See also: [Vault Documentation: Using the App ID auth backend](https://www.vaultproject.io/docs/auth/app-id.html)

**103.3 AppRole authentication**

[AppRole](https://www.vaultproject.io/docs/auth/app-id.html) is intended for machine authentication, like the deprecated (since Vault 0.6.1) [Section 103.2, “AppId authentication”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.authentication.appid). AppRole authentication consists of two hard to guess (secret) tokens: RoleId and SecretId.

Spring Vault supports various AppRole scenarios (push/pull mode and wrapped).

RoleId and optionally SecretId must be provided by configuration, Spring Vault will not look up these or create a custom SecretId.

**Example 103.6. bootstrap.yml with AppRole authentication properties**

spring.cloud.vault:

authentication: APPROLE

app-role:

role-id: bde2076b-cccb-3cf0-d57e-bca7b1e83a52

The following scenarios are supported along the required configuration details:

**Table 103.1. Configuration**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Method** | **RoleId** | **SecretId** | **RoleName** | **Token** |
| Provided RoleId/SecretId | Provided | Provided |  |  |
| Provided RoleId without SecretId | Provided |  |  |  |
| Provided RoleId, Pull SecretId | Provided | Provided | Provided | Provided |
| Pull RoleId, provided SecretId |  | Provided | Provided | Provided |
| Full Pull Mode |  |  | Provided | Provided |
| Wrapped |  |  |  | Provided |
| Wrapped RoleId, provided SecretId | Provided |  |  | Provided |
| Provided RoleId, wrapped SecretId |  | Provided |  | Provided |

**Table 103.2. Pull/Push/Wrapped Matrix**

|  |  |  |
| --- | --- | --- |
| **RoleId** | **SecretId** | **Supported** |
| Provided | Provided | ✅ |
| Provided | Pull | ✅ |
| Provided | Wrapped | ✅ |
| Provided | Absent | ✅ |
| Pull | Provided | ✅ |
| Pull | Pull | ✅ |
| Pull | Wrapped | ❌ |
| Pull | Absent | ❌ |
| Wrapped | Provided | ✅ |
| Wrapped | Pull | ❌ |
| Wrapped | Wrapped | ✅ |
| Wrapped | Absent | ❌ |

|  |
| --- |
| [Note] |
| You can use still all combinations of push/pull/wrapped modes by providing a configured AppRoleAuthentication bean within the bootstrap context. Spring Cloud Vault cannot derive all possible AppRole combinations from the configuration properties. |
| [Important] | **Important** | |
| AppRole authentication is limited to simple pull mode using reactive infrastructure. Full pull mode is not yet supported. Using Spring Cloud Vault with the Spring WebFlux stack enables Vault’s reactive auto-configuration which can be disabled by setting spring.cloud.vault.reactive.enabled=false. | |

**Example 103.7. bootstrap.yml with all AppRole authentication properties**

spring.cloud.vault:

authentication: APPROLE

app-role:

role-id: bde2076b-cccb-3cf0-d57e-bca7b1e83a52

secret-id: 1696536f-1976-73b1-b241-0b4213908d39

role: my-role

app-role-path: approle

* role-id sets the RoleId.
* secret-id sets the SecretId. SecretId can be omitted if AppRole is configured without requiring SecretId (See bind\_secret\_id).
* role: sets the AppRole name for pull mode.
* app-role-path sets the path of the approle authentication mount to use.

See also: [Vault Documentation: Using the AppRole auth backend](https://www.vaultproject.io/docs/auth/approle.html)

**103.4 AWS-EC2 authentication**

The [aws-ec2](https://www.vaultproject.io/docs/auth/aws-ec2.html) auth backend provides a secure introduction mechanism for AWS EC2 instances, allowing automated retrieval of a Vault token. Unlike most Vault authentication backends, this backend does not require first-deploying, or provisioning security-sensitive credentials (tokens, username/password, client certificates, etc.). Instead, it treats AWS as a Trusted Third Party and uses the cryptographically signed dynamic metadata information that uniquely represents each EC2 instance.

**Example 103.8. bootstrap.yml using AWS-EC2 Authentication**

spring.cloud.vault:

authentication: AWS\_EC2

AWS-EC2 authentication enables nonce by default to follow the Trust On First Use (TOFU) principle. Any unintended party that gains access to the PKCS#7 identity metadata can authenticate against Vault.

During the first login, Spring Cloud Vault generates a nonce that is stored in the auth backend aside the instance Id. Re-authentication requires the same nonce to be sent. Any other party does not have the nonce and can raise an alert in Vault for further investigation.

The nonce is kept in memory and is lost during application restart. You can configure a static nonce with spring.cloud.vault.aws-ec2.nonce.

AWS-EC2 authentication roles are optional and default to the AMI. You can configure the authentication role by setting the spring.cloud.vault.aws-ec2.roleproperty.

**Example 103.9. bootstrap.yml with configured role**

spring.cloud.vault:

authentication: AWS\_EC2

aws-ec2:

role: application-server

**Example 103.10. bootstrap.yml with all AWS EC2 authentication properties**

spring.cloud.vault:

authentication: AWS\_EC2

aws-ec2:

role: application-server

aws-ec2-path: aws-ec2

identity-document: http://...

nonce: my-static-nonce

* authentication setting this value to AWS\_EC2 selects the AWS EC2 authentication method
* role sets the name of the role against which the login is being attempted.
* aws-ec2-path sets the path of the AWS EC2 mount to use
* identity-document sets URL of the PKCS#7 AWS EC2 identity document
* nonce used for AWS-EC2 authentication. An empty nonce defaults to nonce generation

See also: [Vault Documentation: Using the aws auth backend](https://www.vaultproject.io/docs/auth/aws.html)

**103.5 AWS-IAM authentication**

The [aws](https://www.vaultproject.io/docs/auth/aws-ec2.html) backend provides a secure authentication mechanism for AWS IAM roles, allowing the automatic authentication with vault based on the current IAM role of the running application. Unlike most Vault authentication backends, this backend does not require first-deploying, or provisioning security-sensitive credentials (tokens, username/password, client certificates, etc.). Instead, it treats AWS as a Trusted Third Party and uses the 4 pieces of information signed by the caller with their IAM credentials to verify that the caller is indeed using that IAM role.

The current IAM role the application is running in is automatically calculated. If you are running your application on AWS ECS then the application will use the IAM role assigned to the ECS task of the running container. If you are running your application naked on top of an EC2 instance then the IAM role used will be the one assigned to the EC2 instance.

When using the AWS-IAM authentication you must create a role in Vault and assign it to your IAM role. An empty role defaults to the friendly name the current IAM role.

**Example 103.11. bootstrap.yml with required AWS-IAM Authentication properties**

spring.cloud.vault:

authentication: AWS\_IAM

**Example 103.12. bootstrap.yml with all AWS-IAM Authentication properties**

spring.cloud.vault:

authentication: AWS\_IAM

aws-iam:

role: my-dev-role

aws-path: aws

server-id: some.server.name

* role sets the name of the role against which the login is being attempted. This should be bound to your IAM role. If one is not supplied then the friendly name of the current IAM user will be used as the vault role.
* aws-path sets the path of the AWS mount to use
* server-id sets the value to use for the X-Vault-AWS-IAM-Server-ID header preventing certain types of replay attacks.

AWS-IAM requires the AWS Java SDK dependency (com.amazonaws:aws-java-sdk-core) as the authentication implementation uses AWS SDK types for credentials and request signing.

See also: [Vault Documentation: Using the aws auth backend](https://www.vaultproject.io/docs/auth/aws.html)

**103.6 TLS certificate authentication**

The cert auth backend allows authentication using SSL/TLS client certificates that are either signed by a CA or self-signed.

To enable cert authentication you need to:

1. Use SSL, see [Chapter 109, *Vault Client SSL configuration*](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.ssl)
2. Configure a Java Keystore that contains the client certificate and the private key
3. Set the spring.cloud.vault.authentication to CERT

**Example 103.13. bootstrap.yml**

spring.cloud.vault:

authentication: CERT

ssl:

key-store: classpath:keystore.jks

key-store-password: changeit

cert-auth-path: cert

See also: [Vault Documentation: Using the Cert auth backend](https://www.vaultproject.io/docs/auth/cert.html)

**103.7 Cubbyhole authentication**

Cubbyhole authentication uses Vault primitives to provide a secured authentication workflow. Cubbyhole authentication uses tokens as primary login method. An ephemeral token is used to obtain a second, login VaultToken from Vault’s Cubbyhole secret backend. The login token is usually longer-lived and used to interact with Vault. The login token will be retrieved from a wrapped response stored at /cubbyhole/response.

**Creating a wrapped token**

|  |
| --- |
| [Note] |
| Response Wrapping for token creation requires Vault 0.6.0 or higher. |

**Example 103.14. Creating and storing tokens**

$ vault token-create -wrap-ttl="10m"

Key Value

--- -----

wrapping\_token: 397ccb93-ff6c-b17b-9389-380b01ca2645

wrapping\_token\_ttl: 0h10m0s

wrapping\_token\_creation\_time: 2016-09-18 20:29:48.652957077 +0200 CEST

wrapped\_accessor: 46b6aebb-187f-932a-26d7-4f3d86a68319

**Example 103.15. bootstrap.yml**

spring.cloud.vault:

authentication: CUBBYHOLE

token: 397ccb93-ff6c-b17b-9389-380b01ca2645

See also:

* [Vault Documentation: Tokens](https://www.vaultproject.io/docs/concepts/tokens.html)
* [Vault Documentation: Cubbyhole Secret Backend](https://www.vaultproject.io/docs/secrets/cubbyhole/index.html)
* [Vault Documentation: Response Wrapping](https://www.vaultproject.io/docs/concepts/response-wrapping.html)

**103.8 Kubernetes authentication**

Kubernetes authentication mechanism (since Vault 0.8.3) allows to authenticate with Vault using a Kubernetes Service Account Token. The authentication is role based and the role is bound to a service account name and a namespace.

A file containing a JWT token for a pod’s service account is automatically mounted at /var/run/secrets/kubernetes.io/serviceaccount/token.

**Example 103.16. bootstrap.yml with all Kubernetes authentication properties**

spring.cloud.vault:

authentication: KUBERNETES

kubernetes:

role: my-dev-role

service-account-token-file: /var/run/secrets/kubernetes.io/serviceaccount/token

* role sets the Role.
* service-account-token-file sets the location of the file containing the Kubernetes Service Account Token. Defaults to /var/run/secrets/kubernetes.io/serviceaccount/token.

See also:

* [Vault Documentation: Kubernetes](https://www.vaultproject.io/docs/auth/kubernetes.html)
* [Kubernetes Documentation: Configure Service Accounts for Pods](https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/)

**104. Secret Backends**

**104.1 Generic Backend**

Spring Cloud Vault supports at the basic level the generic secret backend. The generic secret backend allows storage of arbitrary values as key-value store. A single context can store one or many key-value tuples. Contexts can be organized hierarchically. Spring Cloud Vault allows using the Application name and a default context name (application) in combination with active profiles.

/secret/{application}/{profile}

/secret/{application}

/secret/{default-context}/{profile}

/secret/{default-context}

The application name is determined by the properties:

* spring.cloud.vault.generic.application-name
* spring.cloud.vault.application-name
* spring.application.name

Secrets can be obtained from other contexts within the generic backend by adding their paths to the application name, separated by commas. For example, given the application name usefulapp,mysql1,projectx/aws, each of these folders will be used:

* /secret/usefulapp
* /secret/mysql1
* /secret/projectx/aws

Spring Cloud Vault adds all active profiles to the list of possible context paths. No active profiles will skip accessing contexts with a profile name.

Properties are exposed like they are stored (i.e. without additional prefixes).

spring.cloud.vault:

generic:

enabled: **true**

backend: secret

profile-separator: '/'

default-context: application

application-name: my-app

* enabled setting this value to false disables the secret backend config usage
* backend sets the path of the secret mount to use
* default-context sets the context name used by all applications
* application-name overrides the application name for use in the generic backend
* profile-separator separates the profile name from the context in property sources with profiles

|  |
| --- |
| [Note] |
| The key-value secret backend can be operated in versioned (v2) and non-versioned (v1) modes. Depending on the mode of operation, a different API is required to access secrets. Make sure to enable generic secret backend usage for non-versioned key-value backends and kv secret backend usage for versioned key-value backends. |

See also: [Vault Documentation: Using the KV Secrets Engine - Version 1 (generic secret backend)](https://www.vaultproject.io/docs/secrets/kv/kv-v1.html)

**104.2 Versioned Key-Value Backend**

Spring Cloud Vault supports the versioned Key-Value secret backend. The key-value backend allows storage of arbitrary values as key-value store. A single context can store one or many key-value tuples. Contexts can be organized hierarchically. Spring Cloud Vault allows using the Application name and a default context name (application) in combination with active profiles.

/secret/{application}/{profile}

/secret/{application}

/secret/{default-context}/{profile}

/secret/{default-context}

The application name is determined by the properties:

* spring.cloud.vault.kv.application-name
* spring.cloud.vault.application-name
* spring.application.name

Secrets can be obtained from other contexts within the key-value backend by adding their paths to the application name, separated by commas. For example, given the application name usefulapp,mysql1,projectx/aws, each of these folders will be used:

* /secret/usefulapp
* /secret/mysql1
* /secret/projectx/aws

Spring Cloud Vault adds all active profiles to the list of possible context paths. No active profiles will skip accessing contexts with a profile name.

Properties are exposed like they are stored (i.e. without additional prefixes).

|  |
| --- |
| [Note] |
| Spring Cloud Vault adds the data/ context between the mount path and the actual context path. |

spring.cloud.vault:

kv:

enabled: **true**

backend: secret

profile-separator: '/'

default-context: application

application-name: my-app

* enabled setting this value to false disables the secret backend config usage
* backend sets the path of the secret mount to use
* default-context sets the context name used by all applications
* application-name overrides the application name for use in the generic backend
* profile-separator separates the profile name from the context in property sources with profiles

|  |
| --- |
| [Note] |
| The key-value secret backend can be operated in versioned (v2) and non-versioned (v1) modes. Depending on the mode of operation, a different API is required to access secrets. Make sure to enable generic secret backend usage for non-versioned key-value backends and kv secret backend usage for versioned key-value backends. |

See also: [Vault Documentation: Using the KV Secrets Engine - Version 2 (versioned key-value backend)](https://www.vaultproject.io/docs/secrets/kv/kv-v2.html)

**104.3 Consul**

Spring Cloud Vault can obtain credentials for HashiCorp Consul. The Consul integration requires the spring-cloud-vault-config-consul dependency.

**Example 104.1. pom.xml**

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-vault-config-consul</artifactId>

<version>Finchley.SR1</version>

</dependency>

</dependencies>

The integration can be enabled by setting spring.cloud.vault.consul.enabled=true (default false) and providing the role name with spring.cloud.vault.consul.role=….

The obtained token is stored in spring.cloud.consul.token so using Spring Cloud Consul can pick up the generated credentials without further configuration. You can configure the property name by setting spring.cloud.vault.consul.token-property.

spring.cloud.vault:

consul:

enabled: **true**

role: readonly

backend: consul

token-property: spring.cloud.consul.token

* enabled setting this value to true enables the Consul backend config usage
* role sets the role name of the Consul role definition
* backend sets the path of the Consul mount to use
* token-property sets the property name in which the Consul ACL token is stored

See also: [Vault Documentation: Setting up Consul with Vault](https://www.vaultproject.io/docs/secrets/consul/index.html)

**104.4 RabbitMQ**

Spring Cloud Vault can obtain credentials for RabbitMQ.

The RabbitMQ integration requires the spring-cloud-vault-config-rabbitmq dependency.

**Example 104.2. pom.xml**

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-vault-config-rabbitmq</artifactId>

<version>Finchley.SR1</version>

</dependency>

</dependencies>

The integration can be enabled by setting spring.cloud.vault.rabbitmq.enabled=true (default false) and providing the role name with spring.cloud.vault.rabbitmq.role=….

Username and password are stored in spring.rabbitmq.username and spring.rabbitmq.password so using Spring Boot will pick up the generated credentials without further configuration. You can configure the property names by setting spring.cloud.vault.rabbitmq.username-property andspring.cloud.vault.rabbitmq.password-property.

spring.cloud.vault:

rabbitmq:

enabled: **true**

role: readonly

backend: rabbitmq

username-property: spring.rabbitmq.username

password-property: spring.rabbitmq.password

* enabled setting this value to true enables the RabbitMQ backend config usage
* role sets the role name of the RabbitMQ role definition
* backend sets the path of the RabbitMQ mount to use
* username-property sets the property name in which the RabbitMQ username is stored
* password-property sets the property name in which the RabbitMQ password is stored

See also: [Vault Documentation: Setting up RabbitMQ with Vault](https://www.vaultproject.io/docs/secrets/rabbitmq/index.html)

**104.5 AWS**

Spring Cloud Vault can obtain credentials for AWS.

The AWS integration requires the spring-cloud-vault-config-aws dependency.

**Example 104.3. pom.xml**

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-vault-config-aws</artifactId>

<version>Finchley.SR1</version>

</dependency>

</dependencies>

The integration can be enabled by setting spring.cloud.vault.aws=true (default false) and providing the role name with spring.cloud.vault.aws.role=….

The access key and secret key are stored in cloud.aws.credentials.accessKey and cloud.aws.credentials.secretKey so using Spring Cloud AWS will pick up the generated credentials without further configuration. You can configure the property names by setting spring.cloud.vault.aws.access-key-property andspring.cloud.vault.aws.secret-key-property.

spring.cloud.vault:

aws:

enabled: **true**

role: readonly

backend: aws

access-key-property: cloud.aws.credentials.accessKey

secret-key-property: cloud.aws.credentials.secretKey

* enabled setting this value to true enables the AWS backend config usage
* role sets the role name of the AWS role definition
* backend sets the path of the AWS mount to use
* access-key-property sets the property name in which the AWS access key is stored
* secret-key-property sets the property name in which the AWS secret key is stored

See also: [Vault Documentation: Setting up AWS with Vault](https://www.vaultproject.io/docs/secrets/aws/index.html)

**105. Database backends**

Vault supports several database secret backends to generate database credentials dynamically based on configured roles. This means services that need to access a database no longer need to configure credentials: they can request them from Vault, and use Vault’s leasing mechanism to more easily roll keys.

Spring Cloud Vault integrates with these backends:

* [Section 105.1, “Database”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.backends.database)
* [Section 105.2, “Apache Cassandra”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.backends.cassandra)
* [Section 105.3, “MongoDB”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.backends.mongodb)
* [Section 105.4, “MySQL”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.backends.mysql)
* [Section 105.5, “PostgreSQL”](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#vault.config.backends.postgresql)

Using a database secret backend requires to enable the backend in the configuration and the spring-cloud-vault-config-databases dependency.

Vault ships since 0.7.1 with a dedicated database secret backend that allows database integration via plugins. You can use that specific backend by using the generic database backend. Make sure to specify the appropriate backend path, e.g. spring.cloud.vault.mysql.role.backend=database.

**Example 105.1. pom.xml**

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-vault-config-databases</artifactId>

<version>Finchley.SR1</version>

</dependency>

</dependencies>

|  |
| --- |
| [Note] |
| Enabling multiple JDBC-compliant databases will generate credentials and store them by default in the same property keys hence property names for JDBC secrets need to be configured separately. |

**105.1 Database**

Spring Cloud Vault can obtain credentials for any database listed at <https://www.vaultproject.io/api/secret/databases/index.html>. The integration can be enabled by settingspring.cloud.vault.database.enabled=true (default false) and providing the role name with spring.cloud.vault.database.role=….

While the database backend is a generic one, spring.cloud.vault.database specifically targets JDBC databases. Username and password are stored in spring.datasource.username and spring.datasource.password so using Spring Boot will pick up the generated credentials for your DataSource without further configuration. You can configure the property names by setting spring.cloud.vault.database.username-property andspring.cloud.vault.database.password-property.

spring.cloud.vault:

database:

enabled: **true**

role: readonly

backend: database

username-property: spring.datasource.username

password-property: spring.datasource.username

* enabled setting this value to true enables the Database backend config usage
* role sets the role name of the Database role definition
* backend sets the path of the Database mount to use
* username-property sets the property name in which the Database username is stored
* password-property sets the property name in which the Database password is stored

See also: [Vault Documentation: Database Secrets backend](https://www.vaultproject.io/docs/secrets/databases/index.html)

**105.2 Apache Cassandra**

|  |
| --- |
| [Note] |
| The cassandra backend has been deprecated in Vault 0.7.1 and it is recommended to use the database backend and mount it as cassandra. |

Spring Cloud Vault can obtain credentials for Apache Cassandra. The integration can be enabled by setting spring.cloud.vault.cassandra.enabled=true (default false) and providing the role name with spring.cloud.vault.cassandra.role=….

Username and password are stored in spring.data.cassandra.username and spring.data.cassandra.password so using Spring Boot will pick up the generated credentials without further configuration. You can configure the property names by setting spring.cloud.vault.cassandra.username-property andspring.cloud.vault.cassandra.password-property.

spring.cloud.vault:

cassandra:

enabled: **true**

role: readonly

backend: cassandra

username-property: spring.data.cassandra.username

password-property: spring.data.cassandra.username

* enabled setting this value to true enables the Cassandra backend config usage
* role sets the role name of the Cassandra role definition
* backend sets the path of the Cassandra mount to use
* username-property sets the property name in which the Cassandra username is stored
* password-property sets the property name in which the Cassandra password is stored

See also: [Vault Documentation: Setting up Apache Cassandra with Vault](https://www.vaultproject.io/docs/secrets/cassandra/index.html)

**105.3 MongoDB**

|  |
| --- |
| [Note] |
| The mongodb backend has been deprecated in Vault 0.7.1 and it is recommended to use the database backend and mount it as mongodb. |

Spring Cloud Vault can obtain credentials for MongoDB. The integration can be enabled by setting spring.cloud.vault.mongodb.enabled=true (default false) and providing the role name with spring.cloud.vault.mongodb.role=….

Username and password are stored in spring.data.mongodb.username and spring.data.mongodb.password so using Spring Boot will pick up the generated credentials without further configuration. You can configure the property names by setting spring.cloud.vault.mongodb.username-property andspring.cloud.vault.mongodb.password-property.

spring.cloud.vault:

mongodb:

enabled: **true**

role: readonly

backend: mongodb

username-property: spring.data.mongodb.username

password-property: spring.data.mongodb.password

* enabled setting this value to true enables the MongodB backend config usage
* role sets the role name of the MongoDB role definition
* backend sets the path of the MongoDB mount to use
* username-property sets the property name in which the MongoDB username is stored
* password-property sets the property name in which the MongoDB password is stored

See also: [Vault Documentation: Setting up MongoDB with Vault](https://www.vaultproject.io/docs/secrets/mongodb/index.html)

**105.4 MySQL**

|  |
| --- |
| [Note] |
| The mysql backend has been deprecated in Vault 0.7.1 and it is recommended to use the database backend and mount it as mysql. Configuration for spring.cloud.vault.mysql will be removed in a future version. |

Spring Cloud Vault can obtain credentials for MySQL. The integration can be enabled by setting spring.cloud.vault.mysql.enabled=true (default false) and providing the role name with spring.cloud.vault.mysql.role=….

Username and password are stored in spring.datasource.username and spring.datasource.password so using Spring Boot will pick up the generated credentials without further configuration. You can configure the property names by setting spring.cloud.vault.mysql.username-property andspring.cloud.vault.mysql.password-property.

spring.cloud.vault:

mysql:

enabled: **true**

role: readonly

backend: mysql

username-property: spring.datasource.username

password-property: spring.datasource.username

* enabled setting this value to true enables the MySQL backend config usage
* role sets the role name of the MySQL role definition
* backend sets the path of the MySQL mount to use
* username-property sets the property name in which the MySQL username is stored
* password-property sets the property name in which the MySQL password is stored

See also: [Vault Documentation: Setting up MySQL with Vault](https://www.vaultproject.io/docs/secrets/mysql/index.html)

**105.5 PostgreSQL**

|  |
| --- |
| [Note] |
| The postgresql backend has been deprecated in Vault 0.7.1 and it is recommended to use the database backend and mount it as postgresql. Configuration for spring.cloud.vault.postgresql will be removed in a future version. |

Spring Cloud Vault can obtain credentials for PostgreSQL. The integration can be enabled by setting spring.cloud.vault.postgresql.enabled=true (default false) and providing the role name with spring.cloud.vault.postgresql.role=….

Username and password are stored in spring.datasource.username and spring.datasource.password so using Spring Boot will pick up the generated credentials without further configuration. You can configure the property names by setting spring.cloud.vault.postgresql.username-property andspring.cloud.vault.postgresql.password-property.

spring.cloud.vault:

postgresql:

enabled: **true**

role: readonly

backend: postgresql

username-property: spring.datasource.username

password-property: spring.datasource.username

* enabled setting this value to true enables the PostgreSQL backend config usage
* role sets the role name of the PostgreSQL role definition
* backend sets the path of the PostgreSQL mount to use
* username-property sets the property name in which the PostgreSQL username is stored
* password-property sets the property name in which the PostgreSQL password is stored

See also: [Vault Documentation: Setting up PostgreSQL with Vault](https://www.vaultproject.io/docs/secrets/postgresql/index.html)

**106. Configure PropertySourceLocator behavior**

Spring Cloud Vault uses property-based configuration to create PropertySources for generic and discovered secret backends.

Discovered backends provide VaultSecretBackendDescriptor beans to describe the configuration state to use secret backend as PropertySource. A SecretBackendMetadataFactory is required to create a SecretBackendMetadata object which contains path, name and property transformation configuration.

SecretBackendMetadata is used to back a particular PropertySource.

You can register an arbitrary number of beans implementing VaultConfigurer for customization. Default generic and discovered backend registration is disabled if Spring Cloud Vault discovers at least one VaultConfigurer bean. You can however enable default registration withSecretBackendConfigurer.registerDefaultGenericSecretBackends() and SecretBackendConfigurer.registerDefaultDiscoveredSecretBackends().

**public** **class** CustomizationBean **implements** VaultConfigurer {

*@Override*

**public** **void** addSecretBackends(SecretBackendConfigurer configurer) {

configurer.add("secret/my-application");

configurer.registerDefaultGenericSecretBackends(false);

configurer.registerDefaultDiscoveredSecretBackends(true);

}

}

|  |
| --- |
| [Note] |
| All customization is required to happen in the bootstrap context. Add your configuration classes to META-INF/spring.factories at org.springframework.cloud.bootstrap.BootstrapConfiguration in your application. |

**107. Service Registry Configuration**

You can use a DiscoveryClient (such as from Spring Cloud Consul) to locate a Vault server by setting spring.cloud.vault.discovery.enabled=true (default false). The net result of that is that your apps need a bootstrap.yml (or an environment variable) with the appropriate discovery configuration. The benefit is that the Vault can change its co-ordinates, as long as the discovery service is a fixed point. The default service id is vault but you can change that on the client withspring.cloud.vault.discovery.serviceId.

The discovery client implementations all support some kind of metadata map (e.g. for Eureka we have eureka.instance.metadataMap). Some additional properties of the service may need to be configured in its service registration metadata so that clients can connect correctly. Service registries that do not provide details about transport layer security need to provide a scheme metadata entry to be set either to https or http. If no scheme is configured and the service is not exposed as secure service, then configuration defaults to spring.cloud.vault.scheme which is https when it’s not set.

spring.cloud.vault.discovery:

enabled: **true**

service-id: my-vault-service

**108. Vault Client Fail Fast**

In some cases, it may be desirable to fail startup of a service if it cannot connect to the Vault Server. If this is the desired behavior, set the bootstrap configuration property spring.cloud.vault.fail-fast=true and the client will halt with an Exception.

spring.cloud.vault:

fail-fast: **true**

**109. Vault Client SSL configuration**

SSL can be configured declaratively by setting various properties. You can set either javax.net.ssl.trustStore to configure JVM-wide SSL settings or spring.cloud.vault.ssl.trust-store to set SSL settings only for Spring Cloud Vault Config.

spring.cloud.vault:

ssl:

trust-store: classpath:keystore.jks

trust-store-password: changeit

* trust-store sets the resource for the trust-store. SSL-secured Vault communication will validate the Vault SSL certificate with the specified trust-store.
* trust-store-password sets the trust-store password

Please note that configuring spring.cloud.vault.ssl.\* can be only applied when either Apache Http Components or the OkHttp client is on your class-path.

**110. Lease lifecycle management (renewal and revocation)**

With every secret, Vault creates a lease: metadata containing information such as a time duration, renewability, and more.

Vault promises that the data will be valid for the given duration, or Time To Live (TTL). Once the lease is expired, Vault can revoke the data, and the consumer of the secret can no longer be certain that it is valid.

Spring Cloud Vault maintains a lease lifecycle beyond the creation of login tokens and secrets. That said, login tokens and secrets associated with a lease are scheduled for renewal just before the lease expires until terminal expiry. Application shutdown revokes obtained login tokens and renewable leases.

Secret service and database backends (such as MongoDB or MySQL) usually generate a renewable lease so generated credentials will be disabled on application shutdown.

|  |
| --- |
| [Note] |
| Static tokens are not renewed or revoked. |

Lease renewal and revocation is enabled by default and can be disabled by setting spring.cloud.vault.config.lifecycle.enabled to false. This is not recommended as leases can expire and Spring Cloud Vault cannot longer access Vault or services using generated credentials and valid credentials remain active after application shutdown.

spring.cloud.vault:

config.lifecycle.enabled: **true**

See also: [Vault Documentation: Lease, Renew, and Revoke](https://www.vaultproject.io/docs/concepts/lease.html)

**Part XVI. Spring Cloud Gateway**

**Finchley.SR1**

This project provides an API Gateway built on top of the Spring Ecosystem, including: Spring 5, Spring Boot 2 and Project Reactor. Spring Cloud Gateway aims to provide a simple, yet effective way to route to APIs and provide cross cutting concerns to them such as: security, monitoring/metrics, and resiliency.

**111. How to Include Spring Cloud Gateway**

To include Spring Cloud Gateway in your project use the starter with group org.springframework.cloud and artifact id spring-cloud-starter-gateway. See the [Spring Cloud Project page](https://projects.spring.io/spring-cloud/) for details on setting up your build system with the current Spring Cloud Release Train.

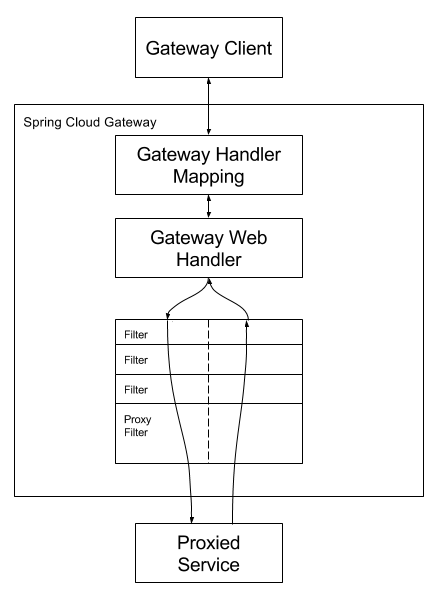
If you include the starter, but, for some reason, you do not want the gateway to be enabled, set spring.cloud.gateway.enabled=false.

|  |  |
| --- | --- |
| [Important] | **Important** |
| Spring Cloud Gateway requires the Netty runtime provided by Spring Boot and Spring Webflux. It does not work in a traditional Servlet Container or built as a WAR. |

**112. Glossary**

* **Route**: Route the basic building block of the gateway. It is defined by an ID, a destination URI, a collection of predicates and a collection of filters. A route is matched if aggregate predicate is true.
* **Predicate**: This is a [Java 8 Function Predicate](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html). The input type is a [Spring Framework ServerWebExchange](https://docs.spring.io/spring/docs/5.0.x/javadoc-api/org/springframework/web/server/ServerWebExchange.html). This allows developers to match on anything from the HTTP request, such as headers or parameters.
* **Filter**: These are instances [Spring Framework GatewayFilter](https://docs.spring.io/spring/docs/5.0.x/javadoc-api/org/springframework/web/server/GatewayFilter.html) constructed in with a specific factory. Here, requests and responses can be modified before or after sending the downstream request.

**113. How It Works**



Clients make requests to Spring Cloud Gateway. If the Gateway Handler Mapping determines that a request matches a Route, it is sent to the Gateway Web Handler. This handler runs sends the request through a filter chain that is specific to the request. The reason the filters are divided by the dotted line, is that filters may execute logic before the proxy request is sent or after. All "pre" filter logic is executed, then the proxy request is made. After the proxy request is made, the "post" filter logic is executed.

|  |
| --- |
| [Note] |
| URIs defined in routes without a port will get a default port set to 80 and 443 for HTTP and HTTPS URIs respectively. |

**114. Route Predicate Factories**

Spring Cloud Gateway matches routes as part of the Spring WebFlux HandlerMapping infrastructure. Spring Cloud Gateway includes many built-in Route Predicate Factories. All of these predicates match on different attributes of the HTTP request. Multiple Route Predicate Factories can be combined and are combined via logical and.

**114.1 After Route Predicate Factory**

The After Route Predicate Factory takes one parameter, a datetime. This predicate matches requests that happen after the current datetime.

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: after\_route

uri: http://example.org

predicates:

- After=2017-01-20T17:42:47.789-07:00[America/Denver**]**

This route matches any request after Jan 20, 2017 17:42 Mountain Time (Denver).

**114.2 Before Route Predicate Factory**

The Before Route Predicate Factory takes one parameter, a datetime. This predicate matches requests that happen before the current datetime.

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: before\_route

uri: http://example.org

predicates:

- Before=2017-01-20T17:42:47.789-07:00[America/Denver**]**

This route matches any request before Jan 20, 2017 17:42 Mountain Time (Denver).

**114.3 Between Route Predicate Factory**

The Between Route Predicate Factory takes two parameters, datetime1 and datetime2. This predicate matches requests that happen after datetime1 and before datetime2. The datetime2 parameter must be after datetime1.

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: between\_route

uri: http://example.org

predicates:

- Between=2017-01-20T17:42:47.789-07:00[America/Denver]**,** 2017-01-21T17:42:47.789-07:00[America/Denver**]**

This route matches any request after Jan 20, 2017 17:42 Mountain Time (Denver) and before Jan 21, 2017 17:42 Mountain Time (Denver). This could be useful for maintenance windows.

**114.4 Cookie Route Predicate Factory**

The Cookie Route Predicate Factory takes two parameters, the cookie name and a regular expression. This predicate matches cookies that have the given name and the value matches the regular expression.

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: cookie\_route

uri: http://example.org

predicates:

- Cookie=chocolate**,** ch.p

This route matches the request has a cookie named chocolate who’s value matches the ch.p regular expression.

**114.5 Header Route Predicate Factory**

The Header Route Predicate Factory takes two parameters, the header name and a regular expression. This predicate matches with a header that has the given name and the value matches the regular expression.

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: header\_route

uri: http://example.org

predicates:

- Header=X-Request-Id**,** \d+

This route matches if the request has a header named X-Request-Id whos value matches the \d+ regular expression (has a value of one or more digits).

**114.6 Host Route Predicate Factory**

The Host Route Predicate Factory takes one parameter: the host name pattern. The pattern is an Ant style pattern with . as the separator. This predicates matches the Host header that matches the pattern.

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: host\_route

uri: http://example.org

predicates:

- Host=\*\*.somehost.org

This route would match if the request has a Host header has the value www.somehost.org or beta.somehost.org.

**114.7 Method Route Predicate Factory**

The Method Route Predicate Factory takes one parameter: the HTTP method to match.

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: method\_route

uri: http://example.org

predicates:

- Method=GET

This route would match if the request method was a GET.

**114.8 Path Route Predicate Factory**

The Path Route Predicate Factory takes one parameter: a Spring PathMatcher pattern.

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: host\_route

uri: http://example.org

predicates:

- Path=/foo/{segment**}**

This route would match if the request path was, for example: /foo/1 or /foo/bar.

This predicate extracts the URI template variables (like segment defined in the example above) as a map of names and values and places it in the ServerWebExchange.getAttributes() with a key defined in PathRoutePredicate.URL\_PREDICATE\_VARS\_ATTR. Those values are then available for use by [GatewayFilter Factories](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#gateway-route-filters)

**114.9 Query Route Predicate Factory**

The Query Route Predicate Factory takes two parameters: a required param and an optional regexp.

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: query\_route

uri: http://example.org

predicates:

- Query=baz

This route would match if the request contained a baz query parameter.

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: query\_route

uri: http://example.org

predicates:

- Query=foo**,** ba.

This route would match if the request contained a foo query parameter whose value matched the ba. regexp, so bar and baz would match.

**114.10 RemoteAddr Route Predicate Factory**

The RemoteAddr Route Predicate Factory takes a list (min size 1) of CIDR-notation (IPv4 or IPv6) strings, e.g. 192.168.0.1/16 (where 192.168.0.1 is an IP address and 16 is a subnet mask).

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: remoteaddr\_route

uri: http://example.org

predicates:

- RemoteAddr=192.168.1.1/24

This route would match if the remote address of the request was, for example, 192.168.1.10.

**114.10.1 Modifying the way remote addresses are resolved**

By default the RemoteAddr Route Predicate Factory uses the remote address from the incoming request. This may not match the actual client IP address if Spring Cloud Gateway sits behind a proxy layer.

You can customize the way that the remote address is resolved by setting a custom RemoteAddressResolver. Spring Cloud Gateway comes with one non-default remote address resolver which is based off of the [X-Forwarded-For header](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Forwarded-For), XForwardedRemoteAddressResolver.

XForwardedRemoteAddressResolver has two static constructor methods which take different approaches to security:

XForwardedRemoteAddressResolver::trustAll returns a RemoteAddressResolver which always takes the first IP address found in the X-Forwarded-Forheader. This approach is vulnerable to spoofing, as a malicious client could set an initial value for the X-Forwarded-For which would be accepted by the resolver.

XForwardedRemoteAddressResolver::maxTrustedIndex takes an index which correlates to the number of trusted infrastructure running in front of Spring Cloud Gateway. If Spring Cloud Gateway is, for example only accessible via HAProxy, then a value of 1 should be used. If two hops of trusted infrastructure are required before Spring Cloud Gateway is accessible, then a value of 2 should be used.

Given the following header value:

X-Forwarded-For: 0.0.0.1, 0.0.0.2, 0.0.0.3

The maxTrustedIndex values below will yield the following remote addresses.

| **maxTrustedIndex** | **result** |
| --- | --- |
| [Integer.MIN\_VALUE,0] | (invalid, IllegalArgumentException during initialization) |
| 1 | 0.0.0.3 |
| 2 | 0.0.0.2 |
| 3 | 0.0.0.1 |
| [4, Integer.MAX\_VALUE] | 0.0.0.1 |

Using Java config:

GatewayConfig.java

RemoteAddressResolver resolver = XForwardedRemoteAddressResolver

.maxTrustedIndex(1);

...

.route("direct-route",

r -> r.remoteAddr("10.1.1.1", "10.10.1.1/24")

.uri("https://downstream1")

.route("proxied-route",

r -> r.remoteAddr(resolver, "10.10.1.1", "10.10.1.1/24")

.uri("https://downstream2")

)

**115. GatewayFilter Factories**

Route filters allow the modification of the incoming HTTP request or outgoing HTTP response in some manner. Route filters are scoped to a particular route. Spring Cloud Gateway includes many built-in GatewayFilter Factories.

NOTE For more detailed examples on how to use any of the following filters, take a look at the [unit tests](https://github.com/spring-cloud/spring-cloud-gateway/tree/master/spring-cloud-gateway-core/src/test/java/org/springframework/cloud/gateway/filter/factory).

**115.1 AddRequestHeader GatewayFilter Factory**

The AddRequestHeader GatewayFilter Factory takes a name and value parameter.

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: add\_request\_header\_route

uri: http://example.org

filters:

- AddRequestHeader=X-Request-Foo**,** Bar

This will add X-Request-Foo:Bar header to the downstream request’s headers for all matching requests.

**115.2 AddRequestParameter GatewayFilter Factory**

The AddRequestParameter GatewayFilter Factory takes a name and value parameter.

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: add\_request\_parameter\_route

uri: http://example.org

filters:

- AddRequestParameter=foo**,** bar

This will add foo=bar to the downstream request’s query string for all matching requests.

**115.3 AddResponseHeader GatewayFilter Factory**

The AddResponseHeader GatewayFilter Factory takes a name and value parameter.

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: add\_request\_header\_route

uri: http://example.org

filters:

- AddResponseHeader=X-Response-Foo**,** Bar

This will add X-Response-Foo:Bar header to the downstream response’s headers for all matching requests.

**115.4 Hystrix GatewayFilter Factory**

[Hystrix](https://github.com/Netflix/Hystrix) is a library from Netflix that implements the [circuit breaker pattern](https://martinfowler.com/bliki/CircuitBreaker.html). The Hystrix GatewayFilter allows you to introduce circuit breakers to your gateway routes, protecting your services from cascading failures and allowing you to provide fallback responses in the event of downstream failures.

To enable Hystrix GatewayFilters in your project, add a dependency on spring-cloud-starter-netflix-hystrix from [Spring Cloud Netflix](https://cloud.spring.io/spring-cloud-netflix/).

The Hystrix GatewayFilter Factory requires a single name parameter, which is the name of the HystrixCommand.

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: hystrix\_route

uri: http://example.org

filters:

- Hystrix=myCommandName

This wraps the remaining filters in a HystrixCommand with command name myCommandName.

The Hystrix filter can also accept an optional fallbackUri parameter. Currently, only forward: schemed URIs are supported. If the fallback is called, the request will be forwarded to the controller matched by the URI.

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: hystrix\_route

uri: lb://backing-service:8088

predicates:

- Path=/consumingserviceendpoint

filters:

- name: Hystrix

args:

name: fallbackcmd

fallbackUri: forward:/incaseoffailureusethis

- RewritePath=/consumingserviceendpoint**,** /backingserviceendpoint

This will forward to the /incaseoffailureusethis URI when the Hystrix fallback is called. Note that this example also demonstrates (optional) Spring Cloud Netflix Ribbon load-balancing via the lb prefix on the destination URI.

Hystrix settings (such as timeouts) can be configured with global defaults or on a route by route basis using application properties as explained on the [Hystrix wiki](https://github.com/Netflix/Hystrix/wiki/Configuration).

To set a 5 second timeout for the example route above, the following configuration would be used:

**application.yml.**

hystrix.command.fallbackcmd.execution.isolation.thread.timeoutInMilliseconds: 5000

**115.5 PrefixPath GatewayFilter Factory**

The PrefixPath GatewayFilter Factory takes a single prefix parameter.

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: prefixpath\_route

uri: http://example.org

filters:

- PrefixPath=/mypath

This will prefix /mypath to the path of all matching requests. So a request to /hello, would be sent to /mypath/hello.

**115.6 PreserveHostHeader GatewayFilter Factory**

The PreserveHostHeader GatewayFilter Factory has not parameters. This filter, sets a request attribute that the routing filter will inspect to determine if the original host header should be sent, rather than the host header determined by the http client.

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: preserve\_host\_route

uri: http://example.org

filters:

- PreserveHostHeader

**115.7 RequestRateLimiter GatewayFilter Factory**

The RequestRateLimiter GatewayFilter Factory is uses a RateLimiter implementation to determine if the current request is allowed to proceed. If it is not, a status of HTTP 429 - Too Many Requests (by default) is returned.

This filter takes an optional keyResolver parameter and parameters specific to the rate limiter (see below).

keyResolver is a bean that implements the KeyResolver interface. In configuration, reference the bean by name using SpEL. #{@myKeyResolver} is a SpEL expression referencing a bean with the name myKeyResolver.

**KeyResolver.java.**

**public** **interface** KeyResolver {

Mono<String> resolve(ServerWebExchange exchange);

}

The KeyResolver interface allows pluggable strategies to derive the key for limiting requests. In future milestones, there will be some KeyResolver implementations.

The default implementation of KeyResolver is the PrincipalNameKeyResolver which retrieves the Principal from the ServerWebExchange and calls Principal.getName().

|  |
| --- |
| [Note] |
| The RequestRateLimiter is not configurable via the "shortcut" notation. The example below is *invalid* |

**application.properties.**

# INVALID SHORTCUT CONFIGURATION

spring.cloud.gateway.routes[0].filters[0]=RequestRateLimiter=2, 2, #{@userkeyresolver}

**115.7.1 Redis RateLimiter**

The redis implementation is based off of work done at [Stripe](https://stripe.com/blog/rate-limiters). It requires the use of the spring-boot-starter-data-redis-reactive Spring Boot starter.

The algorithm used is the [Token Bucket Algorithm](https://en.wikipedia.org/wiki/Token_bucket).

The redis-rate-limiter.replenishRate is how many requests per second do you want a user to be allowed to do, without any dropped requests. This is the rate that the token bucket is filled.

The redis-rate-limiter.burstCapacity is the maximum number of requests a user is allowed to do in a single second. This is the number of tokens the token bucket can hold. Setting this value to zero will block all requests.

A steady rate is accomplished by setting the same value in replenishRate and burstCapacity. Temporary bursts can be allowed by setting burstCapacity higher than replenishRate. In this case, the rate limiter needs to be allowed some time between bursts (according to replenishRate), as 2 consecutive bursts will result in dropped requests (HTTP 429 - Too Many Requests).

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: requestratelimiter\_route

uri: http://example.org

filters:

- name: RequestRateLimiter

args:

redis-rate-limiter.replenishRate: 10

redis-rate-limiter.burstCapacity: 20

**Config.java.**

*@Bean*

KeyResolver userKeyResolver() {

**return** exchange -> Mono.just(exchange.getRequest().getQueryParams().getFirst("user"));

}

This defines a request rate limit of 10 per user. A burst of 20 is allowed, but the next second only 10 requests will be available. The KeyResolver is a simple one that gets the user request parameter (note: this is not recommended for production).

A rate limiter can also be defined as a bean implementing the RateLimiter interface. In configuration, reference the bean by name using SpEL. #{@myRateLimiter}is a SpEL expression referencing a bean with the name myRateLimiter.

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: requestratelimiter\_route

uri: http://example.org

filters:

- name: RequestRateLimiter

args:

rate-limiter: "#{@myRateLimiter}"

key-resolver: "#{@userKeyResolver}"

**115.8 RedirectTo GatewayFilter Factory**

The RedirectTo GatewayFilter Factory takes a status and a url parameter. The status should be a 300 series redirect http code, such as 301. The url should be a valid url. This will be the value of the Location header.

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: prefixpath\_route

uri: http://example.org

filters:

- RedirectTo=302**,** http://acme.org

This will send a status 302 with a Location:http://acme.org header to perform a redirect.

**115.9 RemoveNonProxyHeaders GatewayFilter Factory**

The RemoveNonProxyHeaders GatewayFilter Factory removes headers from forwarded requests. The default list of headers that is removed comes from the [IETF](https://tools.ietf.org/html/draft-ietf-httpbis-p1-messaging-14#section-7.1.3).

**The default removed headers are:**

* Connection
* Keep-Alive
* Proxy-Authenticate
* Proxy-Authorization
* TE
* Trailer
* Transfer-Encoding
* Upgrade

To change this, set the spring.cloud.gateway.filter.remove-non-proxy-headers.headers property to the list of header names to remove.

**115.10 RemoveRequestHeader GatewayFilter Factory**

The RemoveRequestHeader GatewayFilter Factory takes a name parameter. It is the name of the header to be removed.

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: removerequestheader\_route

uri: http://example.org

filters:

- RemoveRequestHeader=X-Request-Foo

This will remove the X-Request-Foo header before it is sent downstream.

**115.11 RemoveResponseHeader GatewayFilter Factory**

The RemoveResponseHeader GatewayFilter Factory takes a name parameter. It is the name of the header to be removed.

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: removeresponseheader\_route

uri: http://example.org

filters:

- RemoveResponseHeader=X-Response-Foo

This will remove the X-Response-Foo header from the response before it is returned to the gateway client.

**115.12 RewritePath GatewayFilter Factory**

The RewritePath GatewayFilter Factory takes a path regexp parameter and a replacement parameter. This uses Java regular expressions for a flexible way to rewrite the request path.

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: rewritepath\_route

uri: http://example.org

predicates:

- Path=/foo/\*\*

filters:

- RewritePath=/foo/(?<segment>.\*)**,** /$\{segment**}**

For a request path of /foo/bar, this will set the path to /bar before making the downstream request. Notice the $\ which is replaced with $ because of the YAML spec.

**115.13 SaveSession GatewayFilter Factory**

The SaveSession GatewayFilter Factory forces a WebSession::save operation *before* forwarding the call downstream. This is of particular use when using something like [Spring Session](https://projects.spring.io/spring-session/) with a lazy data store and need to ensure the session state has been saved before making the forwarded call.

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: save\_session

uri: http://example.org

predicates:

- Path=/foo/\*\*

filters:

- SaveSession

If you are integrating [Spring Security](https://projects.spring.io/spring-security/) with Spring Session, and want to ensure security details have been forwarded to the remote process, this is critical.

**115.14 SecureHeaders GatewayFilter Factory**

The SecureHeaders GatewayFilter Factory adds a number of headers to the response at the reccomendation from [this blog post](https://blog.appcanary.com/2017/http-security-headers.html).

**The following headers are added (allong with default values):**

* X-Xss-Protection:1; mode=block
* Strict-Transport-Security:max-age=631138519
* X-Frame-Options:DENY
* X-Content-Type-Options:nosniff
* Referrer-Policy:no-referrer
* Content-Security-Policy:default-src 'self' https:; font-src 'self' https: data:; img-src 'self' https: data:; object-src 'none'; script-src https:; style-src 'self' https: 'unsafe-inline'
* X-Download-Options:noopen
* X-Permitted-Cross-Domain-Policies:none

To change the default values set the appropriate property in the spring.cloud.gateway.filter.secure-headers namespace:

**Property to change:**

* xss-protection-header
* strict-transport-security
* frame-options
* content-type-options
* referrer-policy
* content-security-policy
* download-options
* permitted-cross-domain-policies

**115.15 SetPath GatewayFilter Factory**

The SetPath GatewayFilter Factory takes a path template parameter. It offers a simple way to manipulate the request path by allowing templated segments of the path. This uses the uri templates from Spring Framework. Multiple matching segments are allowed.

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: setpath\_route

uri: http://example.org

predicates:

- Path=/foo/{segment**}**

filters:

- SetPath=/{segment**}**

For a request path of /foo/bar, this will set the path to /bar before making the downstream request.

**115.16 SetResponseHeader GatewayFilter Factory**

The SetResponseHeader GatewayFilter Factory takes name and value parameters.

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: setresponseheader\_route

uri: http://example.org

filters:

- SetResponseHeader=X-Response-Foo**,** Bar

This GatewayFilter replaces all headers with the given name, rather than adding. So if the downstream server responded with a X-Response-Foo:1234, this would be replaced with X-Response-Foo:Bar, which is what the gateway client would receive.

**115.17 SetStatus GatewayFilter Factory**

The SetStatus GatewayFilter Factory takes a single status parameter. It must be a valid Spring HttpStatus. It may be the integer value 404 or the string representation of the enumeration NOT\_FOUND.

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: setstatusstring\_route

uri: http://example.org

filters:

- SetStatus=BAD\_REQUEST

- id: setstatusint\_route

uri: http://example.org

filters:

- SetStatus=401

In either case, the HTTP status of the response will be set to 401.

**115.18 StripPrefix GatewayFilter Factory**

The StripPrefix GatewayFilter Factory takes one paramter, parts. The parts parameter indicated the number of parts in the path to strip from the request before sending it downstream.

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: nameRoot

uri: http://nameservice

predicates:

- Path=/name/\*\*

filters:

- StripPrefix=2

When a request is made through the gateway to /name/bar/foo the request made to nameservice will look like <http://nameservice/foo>.

**115.19 Retry GatewayFilter Factory**

The Retry GatewayFilter Factory takes retries, statuses, methods, and series as parameters.

* retries: the number of retries that should be attempted
* statuses: the HTTP status codes that should be retried, represented using org.springframework.http.HttpStatus
* methods: the HTTP methods that should be retried, represented using org.springframework.http.HttpMethod
* series: the series of status codes to be retried, represented using org.springframework.http.HttpStatus.Series

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: retry\_test

uri: http://localhost:8080/flakey

predicates:

- Host=\*.retry.com

filters:

- name: Retry

args:

retries: 3

statuses: BAD\_GATEWAY

|  |
| --- |
| [Note] |
| At this time a URI using the forward protocol does not support using the retry filter. |

**116. Global Filters**

The GlobalFilter interface has the same signature as GatewayFilter. These are special filters that are conditionally applied to all routes. (This interface and usage are subject to change in future milestones).

**116.1 Combined Global Filter and GatewayFilter Ordering**

When a request comes in (and matches a Route) the Filtering Web Handler will add all instances of GlobalFilter and all route specific instances of GatewayFilterto a filter chain. This combined filter chain is sorted by the org.springframework.core.Ordered interface, which can be set by implementing the getOrder()method or by using the @Order annotation.

As Spring Cloud Gateway distinguishes between "pre" and "post" phases for filter logic execution (see: How It Works), the filter with the highest precedence will be the first in the "pre"-phase and the last in the "post"-phase.

**ExampleConfiguration.java.**

*@Bean*

*@Order(-1)*

**public** GlobalFilter a() {

**return** (exchange, chain) -> {

log.info("first pre filter");

**return** chain.filter(exchange).then(Mono.fromRunnable(() -> {

log.info("third post filter");

}));

};

}

*@Bean*

*@Order(0)*

**public** GlobalFilter b() {

**return** (exchange, chain) -> {

log.info("second pre filter");

**return** chain.filter(exchange).then(Mono.fromRunnable(() -> {

log.info("second post filter");

}));

};

}

*@Bean*

*@Order(1)*

**public** GlobalFilter c() {

**return** (exchange, chain) -> {

log.info("third pre filter");

**return** chain.filter(exchange).then(Mono.fromRunnable(() -> {

log.info("first post filter");

}));

};

}

**116.2 Forward Routing Filter**

The ForwardRoutingFilter looks for a URI in the exchange attribute ServerWebExchangeUtils.GATEWAY\_REQUEST\_URL\_ATTR. If the url has a forward scheme (ie forward:///localendpoint), it will use the Spring DispatcherHandler to handler the request. The path part of the request URL will be overridden with the path in the forward URL. The unmodified original url is appended to the list in the ServerWebExchangeUtils.GATEWAY\_ORIGINAL\_REQUEST\_URL\_ATTR attribute.

**116.3 LoadBalancerClient Filter**

The LoadBalancerClientFilter looks for a URI in the exchange attribute ServerWebExchangeUtils.GATEWAY\_REQUEST\_URL\_ATTR. If the url has a lb scheme (ie lb://myservice), it will use the Spring Cloud LoadBalancerClient to resolve the name (myservice in the previous example) to an actual host and port and replace the URI in the same attribute. The unmodified original url is appended to the list in the ServerWebExchangeUtils.GATEWAY\_ORIGINAL\_REQUEST\_URL\_ATTR attribute. The filter will also look in the ServerWebExchangeUtils.GATEWAY\_SCHEME\_PREFIX\_ATTR attribute to see if it equals lb and then the same rules apply.

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: myRoute

uri: lb://service

predicates:

- Path=/service/\*\*

**116.4 Netty Routing Filter**

The Netty Routing Filter runs if the url located in the ServerWebExchangeUtils.GATEWAY\_REQUEST\_URL\_ATTR exchange attribute has a http or https scheme. It uses the Netty HttpClient to make the downstream proxy request. The response is put in the ServerWebExchangeUtils.CLIENT\_RESPONSE\_ATTR exchange attribute for use in a later filter. (There is an experimental WebClientHttpRoutingFilter that performs the same function, but does not require netty)

**116.5 Netty Write Response Filter**

The NettyWriteResponseFilter runs if there is a Netty HttpClientResponse in the ServerWebExchangeUtils.CLIENT\_RESPONSE\_ATTR exchange attribute. It is run after all other filters have completed and writes the proxy response back to the gateway client response. (There is an experimental WebClientWriteResponseFilter that performs the same function, but does not require netty)

**116.6 RouteToRequestUrl Filter**

The RouteToRequestUrlFilter runs if there is a Route object in the ServerWebExchangeUtils.GATEWAY\_ROUTE\_ATTR exchange attribute. It creates a new URI, based off of the request URI, but updated with the URI attribute of the Route object. The new URI is placed in the ServerWebExchangeUtils.GATEWAY\_REQUEST\_URL\_ATTR exchange attribute`.

If the URI has a scheme prefix, such as lb:ws://serviceid, the lb scheme is stripped from the URI and placed in the ServerWebExchangeUtils.GATEWAY\_SCHEME\_PREFIX\_ATTR for use later in the filter chain.

**116.7 Websocket Routing Filter**

The Websocket Routing Filter runs if the url located in the ServerWebExchangeUtils.GATEWAY\_REQUEST\_URL\_ATTR exchange attribute has a ws or wss scheme. It uses the Spring Web Socket infrastructure to forward the Websocket request downstream.

Websockets may be load-balanced by prefixing the URI with lb, such as lb:ws://serviceid.

|  |
| --- |
| [Note] |
| If you are using [SockJS](https://github.com/sockjs) as a fallback over normal http, you should configure a normal HTTP route as well as the Websocket Route. |

**application.yml.**

spring:

cloud:

gateway:

routes:

*# SockJS route*

- id: websocket\_sockjs\_route

uri: http://localhost:3001

predicates:

- Path=/websocket/info/\*\*

*# Normwal Websocket route*

- id: websocket\_route

uri: ws://localhost:3001

predicates:

- Path=/websocket/\*\*

**116.8 Gateway Metrics Filter**

To enable Gateway Metrics add spring-boot-starter-actuator as a project dependency. Then, by default, the Gateway Metrics Filter runs as long as the property spring.cloud.gateway.metrics.enabled is not set to false. This filter adds a timer metric named "gateway.requests" with the following tags:

* routeId: The route id
* routeUri: The URI that the API will be routed to
* outcome: Outcome as classified by [HttpStatus.Series](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/HttpStatus.Series.html)
* status: Http Status of the request returned to the client

These metrics are then available to be scraped from /actuator/metrics/gateway.requests and can be easily integated with Prometheus to create a [Grafana](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/images/gateway-grafana-dashboard.jpeg)[dashboard](http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/gateway-grafana-dashboard.json).

|  |
| --- |
| [Note] |
| To enable the pometheus endpoint add micrometer-registry-prometheus as a project dependency. |

**116.9 Making An Exchange As Routed**

After the Gateway has routed a ServerWebExchange it will mark that exchange as "routed" by adding gatewayAlreadyRouted to the exchange attributes. Once a request has been marked as routed, other routing filters will not route the request again, essentially skipping the filter. There are convenience methods that you can use to mark an exchange as routed or check if an exchange has already been routed.

* ServerWebExchangeUtils.isAlreadyRouted takes a ServerWebExchange object and checks if it has been "routed"
* ServerWebExchangeUtils.setAlreadyRouted takes a ServerWebExchange object and marks it as "routed"

**117. TLS / SSL**

The Gateway can listen for requests on https by following the usual Spring server configuration. Example:

**application.yml.**

server:

ssl:

enabled: **true**

key-alias: scg

key-store-password: scg1234

key-store: classpath:scg-keystore.p12

key-store-type: PKCS12

Gateway routes can be routed to both http and https backends. If routing to a https backend then the Gateway can be configured to trust all downstream certificates with the following configuration:

**application.yml.**

spring:

cloud:

gateway:

httpclient:

ssl:

useInsecureTrustManager: **true**

Using an insecure trust manager is not suitable for production. For a production deployment the Gateway can be configured with a set of known certificates that it can trust with the follwing configuration:

**application.yml.**

spring:

cloud:

gateway:

httpclient:

ssl:

trustedX509Certificates:

- cert1.pem

- cert2.pem

**118. Configuration**

Configuration for Spring Cloud Gateway is driven by a collection of `RouteDefinitionLocator`s.

**RouteDefinitionLocator.java.**

**public** **interface** RouteDefinitionLocator {

Flux<RouteDefinition> getRouteDefinitions();

}

By default, a PropertiesRouteDefinitionLocator loads properties using Spring Boot’s @ConfigurationProperties mechanism.

The configuration examples above all use a shortcut notation that uses positional arguments rather than named ones. The two examples below are equivalent:

**application.yml.**

spring:

cloud:

gateway:

routes:

- id: setstatus\_route

uri: http://example.org

filters:

- name: SetStatus

args:

status: 401

- id: setstatusshortcut\_route

uri: http://example.org

filters:

- SetStatus=401

For some usages of the gateway, properties will be adequate, but some production use cases will benefit from loading configuration from an external source, such as a database. Future milestone versions will have RouteDefinitionLocator implementations based off of Spring Data Repositories such as: Redis, MongoDB and Cassandra.

**118.1 Fluent Java Routes API**

To allow for simple configuration in Java, there is a fluent API defined in the RouteLocatorBuilder bean.

**GatewaySampleApplication.java.**

*// static imports from GatewayFilters and RoutePredicates*

*@Bean*

**public** RouteLocator customRouteLocator(RouteLocatorBuilder builder, ThrottleGatewayFilterFactory throttle) {

**return** builder.routes()

.route(r -> r.host("\*\*.abc.org").and().path("/image/png")

.filters(f ->

f.addResponseHeader("X-TestHeader", "foobar"))

.uri("http://httpbin.org:80")

)

.route(r -> r.path("/image/webp")

.filters(f ->

f.addResponseHeader("X-AnotherHeader", "baz"))

.uri("http://httpbin.org:80")

)

.route(r -> r.order(-1)

.host("\*\*.throttle.org").and().path("/get")

.filters(f -> f.filter(throttle.apply(1,

1,

10,

TimeUnit.SECONDS)))

.uri("http://httpbin.org:80")

)

.build();

}

This style also allows for more custom predicate assertions. The predicates defined by RouteDefinitionLocator beans are combined using logical and. By using the fluent Java API, you can use the and(), or() and negate() operators on the Predicate class.

**118.2 DiscoveryClient Route Definition Locator**

The Gateway can be configured to create routes based on services registered with a DiscoveryClient compatible service registry.

To enable this, set spring.cloud.gateway.discovery.locator.enabled=true and make sure a DiscoveryClient implementation is on the classpath and enabled (such as Netflix Eureka, Consul or Zookeeper).

**119. CORS Configuration**

The gateway can be configured to control CORS behavior. The "global" CORS configuration is a map of URL patterns to [Spring Framework CorsConfiguration](https://docs.spring.io/spring/docs/5.0.x/javadoc-api/org/springframework/web/cors/CorsConfiguration.html).

**application.yml.**

spring:

cloud:

gateway:

globalcors:

corsConfigurations:

'[/\*\*]':

allowedOrigins: "docs.spring.io"

allowedMethods:

- GET

In the example above, CORS requests will be allowed from requests that originate from docs.spring.io for all GET requested paths.

**120. Actuator API**

TODO: document the /gateway actuator endpoint

**121. Developer Guide**

TODO: overview of writing custom integrations

**121.1 Writing Custom Route Predicate Factories**

TODO: document writing Custom Route Predicate Factories

**121.2 Writing Custom GatewayFilter Factories**

In order to write a GatewayFilter you will need to implement GatewayFilterFactory. There is an abstract class called AbstractGatewayFilterFactory which you can extend.

**PreGatewayFilterFactory.java.**

**public** **class** PreGatewayFilterFactory **extends** AbstractGatewayFilterFactory<PreGatewayFilterFactory.Config> {

**public** PreGatewayFilterFactory() {

**super**(Config.**class**);

}

*@Override*

**public** GatewayFilter apply(Config config) {

*// grab configuration from Config object*

**return** (exchange, chain) -> {

*//If you want to build a "pre" filter you need to manipulate the*

*//request before calling change.filter*

ServerHttpRequest.Builder builder = exchange.getRequest().mutate();

*//use builder to manipulate the request*

**return** chain.filter(exchange.mutate().request(request).build());

};

}

**public** **static** **class** Config {

*//Put the configuration properties for your filter here*

}

}

**PostGatewayFilterFactory.java.**

**public** **class** PostGatewayFilterFactory **extends** AbstractGatewayFilterFactory<PostGatewayFilterFactory.Config> {

**public** PostGatewayFilterFactory() {

**super**(Config.**class**);

}

*@Override*

**public** GatewayFilter apply(Config config) {

*// grab configuration from Config object*

**return** (exchange, chain) -> {

**return** chain.filter(exchange).then(Mono.fromRunnable(() -> {

ServerHttpResponse response = exchange.getResponse();

*//Manipulate the response in some way*

}));

};

}

**public** **static** **class** Config {

*//Put the configuration properties for your filter here*

}

}

**121.3 Writing Custom Global Filters**

TODO: document writing Custom Global Filters

**121.4 Writing Custom Route Locators and Writers**

TODO: document writing Custom Route Locators and Writers

**122. Building a Simple Gateway Using Spring MVC or Webflux**

Spring Cloud Gateway provides a utility object called ProxyExchange which you can use inside a regular Spring web handler as a method parameter. It supports basic downstream HTTP exchanges via methods that mirror the HTTP verbs. With MVC it also supports forwarding to a local handler via the forward() method. To use the ProxyExchange just include the right module in your classpath (either spring-cloud-gateway-mvc or spring-cloud-gateway-webflux).

MVC example (proxying a request to "/test" downstream to a remote server):

*@RestController*

*@SpringBootApplication*

**public** **class** GatewaySampleApplication {

*@Value("${remote.home}")*

**private** URI home;

*@GetMapping("/test")*

**public** ResponseEntity<?> proxy(ProxyExchange<**byte**[]> proxy) **throws** Exception {

**return** proxy.uri(home.toString() + "/image/png").get();

}

}

The same thing with Webflux:

*@RestController*

*@SpringBootApplication*

**public** **class** GatewaySampleApplication {

*@Value("${remote.home}")*

**private** URI home;

*@GetMapping("/test")*

**public** Mono<ResponseEntity<?>> proxy(ProxyExchange<**byte**[]> proxy) **throws** Exception {

**return** proxy.uri(home.toString() + "/image/png").get();

}

}

There are convenience methods on the ProxyExchange to enable the handler method to discover and enhance the URI path of the incoming request. For example you might want to extract the trailing elements of a path to pass them downstream:

*@GetMapping("/proxy/path/\*\*")*

**public** ResponseEntity<?> proxyPath(ProxyExchange<**byte**[]> proxy) **throws** Exception {

String path = proxy.path("/proxy/path/");

**return** proxy.uri(home.toString() + "/foos/" + path).get();

}

All the features of Spring MVC or Webflux are available to Gateway handler methods. So you can inject request headers and query parameters, for instance, and you can constrain the incoming requests with declarations in the mapping annotation. See the documentation for @RequestMapping in Spring MVC for more details of those features.

Headers can be added to the downstream response using the header() methods on ProxyExchange.

You can also manipulate response headers (and anything else you like in the response) by adding a mapper to the get() etc. method. The mapper is a Function that takes the incoming ResponseEntity and converts it to an outgoing one.

First class support is provided for "sensitive" headers ("cookie" and "authorization" by default) which are not passed downstream, and for "proxy" headers (x-forwarded-\*).

**Part XVII. Spring Cloud Function**

Mark Fisher, Dave Syer

**123. Introduction**

Spring Cloud Function is a project with the following high-level goals:

* Promote the implementation of business logic via functions.
* Decouple the development lifecycle of business logic from any specific runtime target so that the same code can run as a web endpoint, a stream processor, or a task.
* Support a uniform programming model across serverless providers, as well as the ability to run standalone (locally or in a PaaS).
* Enable Spring Boot features (auto-configuration, dependency injection, metrics) on serverless providers.

It abstracts away all of the transport details and infrastructure, allowing the developer to keep all the familiar tools and processes, and focus firmly on business logic.

Here’s a complete, executable, testable Spring Boot application (implementing a simple string manipulation):

*@SpringBootApplication*

**public** **class** Application {

*@Bean*

**public** Function<Flux<String>, Flux<String>> uppercase() {

**return** flux -> flux.map(value -> value.toUpperCase());

}

**public** **static** **void** main(String[] args) {

SpringApplication.run(Application.**class**, args);

}

}

It’s just a Spring Boot application, so it can be built, run and tested, locally and in a CI build, the same way as any other Spring Boot application. The Function is from java.util and Flux is a [Reactive Streams](http://www.reactive-streams.org/) Publisher from [Project Reactor](https://projectreactor.io/). The function can be accessed over HTTP or messaging.

Spring Cloud Function has 4 main features:

1. Wrappers for @Beans of type Function, Consumer and Supplier, exposing them to the outside world as either HTTP endpoints and/or message stream listeners/publishers with RabbitMQ, Kafka etc.
2. Compiling strings which are Java function bodies into bytecode, and then turning them into @Beans that can be wrapped as above.
3. Deploying a JAR file containing such an application context with an isolated classloader, so that you can pack them together in a single JVM.
4. Adapters for [AWS Lambda](https://github.com/spring-cloud/spring-cloud-function/tree/master/spring-cloud-function-adapters/spring-cloud-function-adapter-aws), [Azure](https://github.com/spring-cloud/spring-cloud-function/tree/master/spring-cloud-function-adapters/spring-cloud-function-adapter-azure), [Apache OpenWhisk](https://github.com/spring-cloud/spring-cloud-function/tree/master/spring-cloud-function-adapters/spring-cloud-function-adapter-openwhisk) and possibly other "serverless" service providers.

|  |
| --- |
| [Note] |
| Spring Cloud is released under the non-restrictive Apache 2.0 license. If you would like to contribute to this section of the documentation or if you find an error, please find the source code and issue trackers in the project at [github](https://github.com/spring-cloud/spring-cloud-function/tree/master/docs/src/main/asciidoc). |

**124. Getting Started**

Build from the command line (and "install" the samples):

$ ./mvnw clean install

(If you like to YOLO add -DskipTests.)

Run one of the samples, e.g.

$ java -jar spring-cloud-function-samples/function-sample/target/\*.jar

This runs the app and exposes its functions over HTTP, so you can convert a string to uppercase, like this:

$ curl -H "Content-Type: text/plain" localhost:8080/uppercase -d Hello

HELLO

You can convert multiple strings (a Flux<String>) by separating them with new lines

$ curl -H "Content-Type: text/plain" localhost:8080/uppercase -d 'Hello

> World'

HELLOWORLD

(You can use QJ in a terminal to insert a new line in a literal string like that.)

**125. Building and Running a Function**

The sample @SpringBootApplication above has a function that can be decorated at runtime by Spring Cloud Function to be an HTTP endpoint, or a Stream processor, for instance with RabbitMQ, Apache Kafka or JMS.

The @Beans can be Function, Consumer or Supplier (all from java.util), and their parametric types can be String or POJO. A Function is exposed as a Spring Cloud Stream Processor if spring-cloud-function-stream is on the classpath. A Consumer is also exposed as a Stream Sink and a Suppliertranslates to a Stream Source. HTTP endpoints are exposed if the Stream binder is spring-cloud-stream-binder-servlet.

Functions can be of Flux<String> or Flux<Pojo> and Spring Cloud Function takes care of converting the data to and from the desired types, as long as it comes in as plain text or (in the case of the POJO) JSON. TBD: support for Flux<Message<Pojo>> and maybe plain Pojo types (Fluxes implied and implemented by the framework).

Functions can be grouped together in a single application, or deployed one-per-jar. It’s up to the developer to choose. An app with multiple functions can be deployed multiple times in different "personalities", exposing different functions over different physical transports.

**126. Function Catalog and Flexible Function Signatures**

One of the main features of Spring Cloud Function is to adapt and support a range of type signatures for user-defined functions. So users can supply a bean of type Function<String,String>, for instance, and the FunctionCatalog will wrap it into a Function<Flux<String>,Flux<String>>. Users don’t normally have to care about the FunctionCatalog at all, but it is useful to know what kind of functions are supported in user code.

Generally speaking users can expect that if they write a function for a plain old Java type (or primitive wrapper), then the function catalog will wrap it to a Flux of the same type. If the user writes a function using Message (from spring-messaging) it will receive and transmit headers from any adapter that supports key-value metadata (e.g. HTTP headers). Here are the details.

| **User Function** | **Catalog Registration** |  |
| --- | --- | --- |
| Function<S,T> | Function<Flux<S>, Flux<T>> |  |
| Function<Message<S>,Message<T>> | Function<Flux<Message<S>>, Flux<Message<T>>> |  |
| Function<Flux<S>, Flux<T>> | Function<Flux<S>, Flux<T>> (pass through) |  |
| Supplier<T> | Supplier<Flux<T>> |  |
| Supplier<Flux<T>> | Supplier<Flux<T>> |  |
| Consumer<T> | Function<Flux<T>, Mono<Void>> |  |
| Consumer<Message<T>> | Function<Flux<Message<T>>, Mono<Void>> |  |
| Consumer<Flux<T>> | Consumer<Flux<T>> |  |

Consumer is a little bit special because it has a void return type, which implies blocking, at least potentially. Most likely you will not need to write Consumer<Flux<?>>, but if you do need to do that, remember to subscribe to the input flux. If you declare a Consumer of a non publisher type (which is normal), it will be converted to a function that returns a publisher, so that it can be subscribed to in a controlled way.

A function catalog can contain a Supplier and a Function (or Consumer) with the same name (like a GET and a POST to the same resource). It can even contain a Consumer<Flux<>> with the same name as a Function, but it cannot contain a Consumer<T> and a Function<T,S> with the same name when T is not a Publisher because the consumer would be converted to a Function and only one of them can be registered.

**127. Standalone Web Applications**

The spring-cloud-function-web module has autoconfiguration that activates when it is included in a Spring Boot web application (with MVC support). There is also a spring-cloud-starter-function-web to collect all the optional dependnecies in case you just want a simple getting started experience.

With the web configurations activated your app will have an MVC endpoint (on "/" by default, but configurable with spring.cloud.function.web.path) that can be used to access the functions in the application context. The supported content types are plain text and JSON.

| **Method** | **Path** | **Request** | **Response** | **Status** |
| --- | --- | --- | --- | --- |
| GET | /{supplier} | - | Items from the named supplier | 200 OK |
| POST | /{consumer} | JSON object or text | Mirrors input and pushes request body into consumer | 202 Accepted |
| POST | /{consumer} | JSON array or text with new lines | Mirrors input and pushes body into consumer one by one | 202 Accepted |
| POST | /{function} | JSON object or text | The result of applying the named function | 200 OK |
| POST | /{function} | JSON array or text with new lines | The result of applying the named function | 200 OK |
| GET | /{function}/{item} | - | Convert the item into an object and return the result of applying the function | 200 OK |

As the table above shows the behaviour of the endpoint depends on the method and also the type of incoming request data. When the incoming data is single valued, and the target function is declared as obviously single valued (i.e. not returning a collection or Flux), then the response will also contain a single value. For multi-valued responses the client can ask for a server-sent event stream by sending `Accept: text/event-stream". If there is only one function (consumer etc.) then the name in the path is optional. Composite functions can be addressed using pipes or commas to separate function names (pipes are legal in URL paths, but a bit awkward to type on the command line).

Functions and consumers that are declared with input and output in Message<?> will see the request headers on the input messages, and the output message headers will be converted to HTTP headers.

When POSTing text the response format might be different with Spring Boot 2.0 and older versions, depending on the content negotiation (provide content type and accpt headers for the best results).

**128. Standalone Streaming Applications**

To send or receive messages from a broker (such as RabbitMQ or Kafka) you can use the spring-cloud-function-stream adapter. Add the adapter to your classpath along with the appropriate binder from Spring Cloud Stream. The adapter will bind to the message broker as a Processor (input and output streams) unless the user explicitly disables one or the other using spring.cloud.function.stream.{source,sink}.enabled=false.

An incoming message is routed to a function (or consumer). If there is only one, then the choice is obvious. If there are multiple functions that can accept an incoming message, the message is inspected to see if there is a stream\_routekey header containing the name of a function. Routing headers or function names can be composed using a comma- or pipe-separated name. The header is also added to outgoing messages from a supplier. Messages with no route key can be routed exclusively to a function or consumer by specifying spring.cloud.function.stream.{processor,sink}.name. If a single function cannot be identified to process an incoming message there will be an error, unless you set spring.cloud.function.stream.shared=true, in which case such messages will be sent to all compatible functions. A single supplier can be chosen for output messages from a supplier (if more than one is available) using the spring.cloud.function.stream.source.name.

|  |
| --- |
| [Note] |
| some binders will fail on startup if the message broker is not available and the function catalog contains suppliers that immediately produce messages when accessed. You can switch off the automatic publishing from suppliers on startup using the spring.cloud.function.strean.supplier.enabled=falseflag. |

**129. Deploying a Packaged Function**

Spring Cloud Function provides a "deployer" library that allows you to launch a jar file (or exploded archive, or set of jar files) with an isolated class loader and expose the functions defined in it. This is quite a powerful tool that would allow you to, for instance, adapt a function to a range of different input-output adapters without changing the target jar file. Serverless platforms often have this kind of feature built in, so you could see it as a building block for a function invoker in such a platform (indeed the [Riff](https://projectriff.io/)Java function invoker uses this library).

The standard entry point of the API is the Spring configuration annotation @EnableFunctionDeployer. If that is used in a Spring Boot application the deployer kicks in and looks for some configuration to tell it where to find the function jar. At a minimum the user has to provide a function.location which is a URL or resource location for the archive containing the functions. It can optionally use a maven: prefix to locate the artifact via a dependency lookup (see FunctionProperties for complete details). A Spring Boot application is bootstrapped from the jar file, using the MANIFEST.MF to locate a start class, so that a standard Spring Boot fat jar works well, for example. If the target jar can be launched successfully then the result is a function registered in the main application’s FunctionCatalog. The registered function can be applied by code in the main application, even though it was created in an isolated class loader (by deault).

**130. Dynamic Compilation**

There is a sample app that uses the function compiler to create a function from a configuration property. The vanilla "function-sample" also has that feature. And there are some scripts that you can run to see the compilation happening at run time. To run these examples, change into the scripts directory:

cd scripts

Also, start a RabbitMQ server locally (e.g. execute rabbitmq-server).

Start the Function Registry Service:

./function-registry.sh

Register a Function:

./registerFunction.sh -n uppercase -f "f->f.map(s->s.toString().toUpperCase())"

Run a REST Microservice using that Function:

./web.sh -f uppercase -p 9000

curl -H "Content-Type: text/plain" -H "Accept: text/plain" localhost:9000/uppercase -d foo

Register a Supplier:

./registerSupplier.sh -n words -f "()->Flux.just(\"foo\",\"bar\")"

Run a REST Microservice using that Supplier:

./web.sh -s words -p 9001

curl -H "Accept: application/json" localhost:9001/words

Register a Consumer:

./registerConsumer.sh -n print -t String -f "System.out::println"

Run a REST Microservice using that Consumer:

./web.sh -c print -p 9002

curl -X POST -H "Content-Type: text/plain" -d foo localhost:9002/print

Run Stream Processing Microservices:

First register a streaming words supplier:

./registerSupplier.sh -n wordstream -f "()->Flux.interval(Duration.ofMillis(1000)).map(i->\"message-\"+i)"

Then start the source (supplier), processor (function), and sink (consumer) apps (in reverse order):

./stream.sh -p 9103 -i uppercaseWords -c print

./stream.sh -p 9102 -i words -f uppercase -o uppercaseWords

./stream.sh -p 9101 -s wordstream -o words

The output will appear in the console of the sink app (one message per second, converted to uppercase):

MESSAGE-0

MESSAGE-1

MESSAGE-2

MESSAGE-3

MESSAGE-4

MESSAGE-5

MESSAGE-6

MESSAGE-7

MESSAGE-8

MESSAGE-9

...

**131. Serverless Platform Adapters**

As well as being able to run as a standalone process, a Spring Cloud Function application can be adapted to run one of the existing serverless platforms. In the project there are adapters for [AWS Lambda](https://github.com/spring-cloud/spring-cloud-function/tree/master/spring-cloud-function-adapters/spring-cloud-function-adapter-aws), [Azure](https://github.com/spring-cloud/spring-cloud-function/tree/master/spring-cloud-function-adapters/spring-cloud-function-adapter-azure), and [Apache OpenWhisk](https://github.com/spring-cloud/spring-cloud-function/tree/master/spring-cloud-function-adapters/spring-cloud-function-adapter-openwhisk). The [Oracle Fn platform](https://github.com/fnproject/fn) has its own Spring Cloud Function adapter. And [Riff](https://projectriff.io/) supports Java functions and its [Java Function Invoker](https://github.com/projectriff/java-function-invoker) acts natively is an adapter for Spring Cloud Function jars.

**131.1 AWS Lambda**

The [AWS](https://aws.amazon.com/) adapter takes a Spring Cloud Function app and converts it to a form that can run in AWS Lambda.

**131.1.1 Introduction**

The adapter has a couple of generic request handlers that you can use. The most generic is SpringBootStreamHandler, which uses a Jackson ObjectMapperprovided by Spring Boot to serialize and deserialize the objects in the function. There is also a SpringBootRequestHandler which you can extend, and provide the input and output types as type parameters (enabling AWS to inspect the class and do the JSON conversions itself).

If your app has more than one @Bean of type Function etc. then you can choose the one to use by configuring function.name (e.g. as FUNCTION\_NAMEenvironment variable in AWS). The functions are extracted from the Spring Cloud FunctionCatalog (searching first for Function then Consumer and finally Supplier).

**131.1.2 Notes on JAR Layout**

You don’t need the Spring Cloud Function Web or Stream adapter at runtime in Lambda, so you might need to exclude those before you create the JAR you send to AWS. A Lambda application has to be shaded, but a Spring Boot standalone application does not, so you can run the same app using 2 separate jars (as per the sample). The sample app creates 2 jar files, one with an aws classifier for deploying in Lambda, and one executable (thin) jar that includes spring-cloud-function-web at runtime. Spring Cloud Function will try and locate a "main class" for you from the JAR file manifest, using the Start-Class attribute (which will be added for you by the Spring Boot tooling if you use the starter parent). If there is no Start-Class in your manifest you can use an environment variable MAIN\_CLASS when you deploy the function to AWS.

**131.1.3 Upload**

Build the sample under spring-cloud-function-samples/function-sample-aws and upload the -aws jar file to Lambda. The handler can be example.Handleror org.springframework.cloud.function.adapter.aws.SpringBootStreamHandler (FQN of the class, *not* a method reference, although Lambda does accept method references).

./mvnw -U clean package

Using the AWS command line tools it looks like this:

aws lambda create-function --function-name Uppercase --role arn:aws:iam::[USERID]:role/service-role/[ROLE] --zip-file fileb://function-sample-aws/target/function-sample-aws-1.0.0.RELEASE-aws.jar --handler org.springframework.cloud.function.adapter.aws.SpringBootStreamHandler --description "Spring Cloud Function Adapter Example" --runtime java8 --region us-east-1 --timeout 30 --memory-size 1024 --publish

The input type for the function in the AWS sample is a Foo with a single property called "value". So you would need this to test it:

{

"value": "test"

}

**131.1.4 Platfom Specific Features**

**HTTP and API Gateway**

AWS has some platform-specific data types, including batching of messages, which is much more efficient than processing each one individually. To make use of these types you can write a function that depends on those types. Or you can rely on Spring to extract the data from the AWS types and convert it to a Spring Message. To do this you tell AWS that the function is of a specific generic handler type (depending on the AWS service) and provide a bean of type Function<Message<S>,Message<T>>, where S and T are your business data types. If there is more than one bean of type Function you may also need to configure the Spring Boot property function.name to be the name of the target bean (e.g. use FUNCTION\_NAME as an environment variable).

The supported AWS services and generic handler types are listed below:

| **Service** | **AWS Types** | **Generic Handler** |  |
| --- | --- | --- | --- |
| API Gateway | APIGatewayProxyRequestEvent, APIGatewayProxyResponseEvent | org.springframework.cloud.function.adapter.aws.SpringBootApiGatewayRequestHandler |  |
| Kinesis | KinesisEvent | org.springframework.cloud.function.adapter.aws.SpringBootKinesisEventHandler |  |

For example, to deploy behind an API Gateway, use --handler org.springframework.cloud.function.adapter.aws.SpringBootApiGatewayRequestHandler in your AWS command line (in via the UI) and define a @Bean of type Function<Message<Foo>,Message<Bar>> where Foo and Bar are POJO types (the data will be marshalled and unmarshalled by AWS using Jackson).

**131.2 Azure Functions**

The [Azure](https://azure.microsoft.com/) adapter bootstraps a Spring Cloud Function context and channels function calls from the Azure framework into the user functions, using Spring Boot configuration where necessary. Azure Functions has quite a unique, but invasive programming model, involving annotations in user code that are specific to the platform. The Spring Cloud Function Azure adapter trades the convenience of these annotations for portability of the function implementations. Instead of using the annotations you have to write some JSON by hand (at least for now) to guide the platform to call the right methods in the adapter.

This project provides an adapter layer for a Spring Cloud Function application onto Azure. You can write an app with a single @Bean of type Function and it will be deployable in Azure if you get the JAR file laid out right.

The adapter has a generic HTTP request handler that you can use optionally. There is a AzureSpringBootRequestHandler which you must extend, and provide the input and output types as type parameters (enabling Azure to inspect the class and do the JSON conversions itself).

If your app has more than one @Bean of type Function etc. then you can choose the one to use by configuring function.name. The functions are extracted from the Spring Cloud FunctionCatalog.

**131.2.1 Notes on JAR Layout**

You don’t need the Spring Cloud Function Web at runtime in Azure, so you need to exclude this before you create the JAR you deploy to Azure. A function application on Azure has to be shaded, but a Spring Boot standalone application does not, so you can run the same app using 2 separate jars (as per the sample here). The sample app creates the shaded jar file, with an azure classifier for deploying in Azure.

**131.2.2 JSON Configuration**

The Azure tooling needs to find some JSON configuration files to tell it how to deploy and integrate the function (e.g. which Java class to use as the entry point, and which triggers to use). Those files can be created with the Maven plugin for a non-Spring function, but the tooling doesn’t work yet with the adapter in its current form. There is an example function.json in the sample which hooks the function up as an HTTP endpoint:

{

"scriptFile" : "../function-sample-azure-1.0.0.RELEASE-azure.jar",

"entryPoint" : "example.FooHandler.execute",

"bindings" : [ {

"type" : "httpTrigger",

"name" : "foo",

"direction" : "in",

"authLevel" : "anonymous",

"methods" : [ "get", "post" ]

}, {

"type" : "http",

"name" : "$return",

"direction" : "out"

} ],

"disabled" : false

}

**131.2.3 Build**

./mvnw -U clean package

**131.2.4 Running the sample**

You can run the sample locally, just like the other Spring Cloud Function samples:

and curl -H "Content-Type: text/plain" localhost:8080/function -d '{"value": "hello foobar"}'.

You will need the az CLI app and some node.js fu (see <https://docs.microsoft.com/en-us/azure/azure-functions/functions-create-first-java-maven> for more detail). To deploy the function on Azure runtime:

$ az login

$ mvn azure-functions:deploy

On another terminal try this: curl https://<azure-function-url-from-the-log>/api/uppercase -d '{"value": "hello foobar!"}'. Please ensure that you use the right URL for the function above. Alternatively you can test the function in the Azure Dashboard UI (click on the function name, go to the right hand side and click "Test" and to the bottom right, "Run").

The input type for the function in the Azure sample is a Foo with a single property called "value". So you need this to test it with something like below:

{

"value": "foobar"

}

**131.3 Apache Openwhisk**

The [OpenWhisk](https://openwhisk.apache.org/) adapter is in the form of an executable jar that can be used in a a docker image to be deployed to Openwhisk. The platform works in request-response mode, listening on port 8080 on a specific endpoint, so the adapter is a simple Spring MVC application.

**131.3.1 Quick Start**

Implement a POF (be sure to use the functions package):

package functions;

import java.util.function.Function;

public class Uppercase implements Function<String, String> {

public String apply(String input) {

return input.toUpperCase();

}

}

Install it into your local Maven repository:

./mvnw clean install

Create a function.properties file that provides its Maven coordinates. For example:

dependencies.function: com.example:pof:0.0.1-SNAPSHOT

Copy the openwhisk runner JAR to the working directory (same directory as the properties file):

cp spring-cloud-function-adapters/spring-cloud-function-adapter-openwhisk/target/spring-cloud-function-adapter-openwhisk-1.0.0.RELEASE.jar runner.jar

Generate a m2 repo from the --thin.dryrun of the runner JAR with the above properties file:

java -jar -Dthin.root=m2 runner.jar --thin.name=function --thin.dryrun

Use the following Dockerfile:

FROM openjdk:8-jdk-alpine

VOLUME /tmp

COPY m2 /m2

ADD runner.jar .

ADD function.properties .

ENV JAVA\_OPTS=""

ENTRYPOINT [ "java", "-Djava.security.egd=file:/dev/./urandom", "-jar", "runner.jar", "--thin.root=/m2", "--thin.name=function", "--function.name=uppercase"]

EXPOSE 8080

|  |
| --- |
| [Note] |
| you could use a Spring Cloud Function app, instead of just a jar with a POF in it, in which case you would have to change the way the app runs in the container so that it picks up the main class as a source file. For example, you could change the ENTRYPOINT above and add --spring.main.sources=com.example.SampleApplication. |

Build the Docker image:

docker build -t [username/appname] .

Push the Docker image:

docker push [username/appname]

Use the OpenWhisk CLI (e.g. after vagrant ssh) to create the action:

wsk action create example --docker [username/appname]

Invoke the action:

wsk action invoke example --result --param payload foo

{

"result": "FOO"

}

**Part XVIII. Appendix: Compendium of Configuration Properties**

| **Name** | **Default** | **Description** |
| --- | --- | --- |
| encrypt.fail-on-error | true | Flag to say that a process should fail if there is an encryption or decryption error. |
| encrypt.key |  | A symmetric key. As a stronger alternative consider using a keystore. |
| encrypt.key-store.alias |  | Alias for a key in the store. |
| encrypt.key-store.location |  | Location of the key store file, e.g. classpath:/keystore.jks. |
| encrypt.key-store.password |  | Password that locks the keystore. |
| encrypt.key-store.secret |  | Secret protecting the key (defaults to the same as the password). |
| encrypt.rsa.algorithm |  | The RSA algorithm to use (DEFAULT or OEAP). Once it is set do not change it (or existing ciphers will not a decryptable). |
| encrypt.rsa.salt | deadbeef | Salt for the random secret used to encrypt cipher text. Once it is set do not change it (or existing ciphers will not a decryptable). |
| encrypt.rsa.strong | false | Flag to indicate that "strong" AES encryption should be used internally. If true then the GCM algorithm is applied to the AES encrypted bytes. Default is false (in which case "standard" CBC is used instead). Once it is set do not change it (or existing ciphers will not a decryptable). |
| encrypt.salt | deadbeef | A salt for the symmetric key in the form of a hex-encoded byte array. As a stronger alternative consider using a keystore. |
| endpoints.zookeeper.enabled | true | Enable the /zookeeper endpoint to inspect the state of zookeeper. |
| eureka.client.allow-redirects | false | Indicates whether server can redirect a client request to a backup server/cluster. If set to false, the server will handle the request directly, If set to true, it may send HTTP redirect to the client, with a new server location. |
| eureka.client.availability-zones |  | Gets the list of availability zones (used in AWS data centers) for the region in which this instance resides. The changes are effective at runtime at the next registry fetch cycle as specified by registryFetchIntervalSeconds. |
| eureka.client.backup-registry-impl |  | Gets the name of the implementation which implements BackupRegistry to fetch the registry information as a fall back option for only the first time when the eureka client starts. This may be needed for applications which needs additional resiliency for registry information without which it cannot operate. |
| eureka.client.cache-refresh-executor-exponential-back-off-bound | 10 | Cache refresh executor exponential back off related property. It is a maximum multiplier value for retry delay, in case where a sequence of timeouts occurred. |
| eureka.client.cache-refresh-executor-thread-pool-size | 2 | The thread pool size for the cacheRefreshExecutor to initialise with |
| eureka.client.client-data-accept |  | EurekaAccept name for client data accept |
| eureka.client.decoder-name |  | This is a transient config and once the latest codecs are stable, can be removed (as there will only be one) |
| eureka.client.disable-delta | false | Indicates whether the eureka client should disable fetching of delta and should rather resort to getting the full registry information. Note that the delta fetches can reduce the traffic tremendously, because the rate of change with the eureka server is normally much lower than the rate of fetches. The changes are effective at runtime at the next registry fetch cycle as specified by registryFetchIntervalSeconds |
| eureka.client.dollar-replacement | \_- | Get a replacement string for Dollar sign <code>$</code> during serializing/deserializing information in eureka server. |
| eureka.client.enabled | true | Flag to indicate that the Eureka client is enabled. |
| eureka.client.encoder-name |  | This is a transient config and once the latest codecs are stable, can be removed (as there will only be one) |
| eureka.client.escape-char-replacement | \_\_ | Get a replacement string for underscore sign <code>\_</code> during serializing/deserializing information in eureka server. |
| eureka.client.eureka-connection-idle-timeout-seconds | 30 | Indicates how much time (in seconds) that the HTTP connections to eureka server can stay idle before it can be closed. In the AWS environment, it is recommended that the values is 30 seconds or less, since the firewall cleans up the connection information after a few mins leaving the connection hanging in limbo |
| eureka.client.eureka-server-connect-timeout-seconds | 5 | Indicates how long to wait (in seconds) before a connection to eureka server needs to timeout. Note that the connections in the client are pooled by org.apache.http.client.HttpClient and this setting affects the actual connection creation and also the wait time to get the connection from the pool. |
| eureka.client.eureka-server-d-n-s-name |  | Gets the DNS name to be queried to get the list of eureka servers.This information is not required if the contract returns the service urls by implementing serviceUrls. The DNS mechanism is used when useDnsForFetchingServiceUrls is set to true and the eureka client expects the DNS to configured a certain way so that it can fetch changing eureka servers dynamically. The changes are effective at runtime. |
| eureka.client.eureka-server-port |  | Gets the port to be used to construct the service url to contact eureka server when the list of eureka servers come from the DNS.This information is not required if the contract returns the service urls eurekaServerServiceUrls(String). The DNS mechanism is used when useDnsForFetchingServiceUrls is set to true and the eureka client expects the DNS to configured a certain way so that it can fetch changing eureka servers dynamically. The changes are effective at runtime. |
| eureka.client.eureka-server-read-timeout-seconds | 8 | Indicates how long to wait (in seconds) before a read from eureka server needs to timeout. |
| eureka.client.eureka-server-total-connections | 200 | Gets the total number of connections that is allowed from eureka client to all eureka servers. |
| eureka.client.eureka-server-total-connections-per-host | 50 | Gets the total number of connections that is allowed from eureka client to a eureka server host. |
| eureka.client.eureka-server-u-r-l-context |  | Gets the URL context to be used to construct the service url to contact eureka server when the list of eureka servers come from the DNS. This information is not required if the contract returns the service urls from eurekaServerServiceUrls. The DNS mechanism is used when useDnsForFetchingServiceUrls is set to true and the eureka client expects the DNS to configured a certain way so that it can fetch changing eureka servers dynamically. The changes are effective at runtime. |
| eureka.client.eureka-service-url-poll-interval-seconds | 0 | Indicates how often(in seconds) to poll for changes to eureka server information. Eureka servers could be added or removed and this setting controls how soon the eureka clients should know about it. |
| eureka.client.fetch-registry | true | Indicates whether this client should fetch eureka registry information from eureka server. |
| eureka.client.fetch-remote-regions-registry |  | Comma separated list of regions for which the eureka registry information will be fetched. It is mandatory to define the availability zones for each of these regions as returned by availabilityZones. Failing to do so, will result in failure of discovery client startup. |
| eureka.client.filter-only-up-instances | true | Indicates whether to get the applications after filtering the applications for instances with only InstanceStatus UP states. |
| eureka.client.g-zip-content | true | Indicates whether the content fetched from eureka server has to be compressed whenever it is supported by the server. The registry information from the eureka server is compressed for optimum network traffic. |
| eureka.client.heartbeat-executor-exponential-back-off-bound | 10 | Heartbeat executor exponential back off related property. It is a maximum multiplier value for retry delay, in case where a sequence of timeouts occurred. |
| eureka.client.heartbeat-executor-thread-pool-size | 2 | The thread pool size for the heartbeatExecutor to initialise with |
| eureka.client.initial-instance-info-replication-interval-seconds | 40 | Indicates how long initially (in seconds) to replicate instance info to the eureka server |
| eureka.client.instance-info-replication-interval-seconds | 30 | Indicates how often(in seconds) to replicate instance changes to be replicated to the eureka server. |
| eureka.client.log-delta-diff | false | Indicates whether to log differences between the eureka server and the eureka client in terms of registry information. Eureka client tries to retrieve only delta changes from eureka server to minimize network traffic. After receiving the deltas, eureka client reconciles the information from the server to verify it has not missed out some information. Reconciliation failures could happen when the client has had network issues communicating to server.If the reconciliation fails, eureka client gets the full registry information. While getting the full registry information, the eureka client can log the differences between the client and the server and this setting controls that. The changes are effective at runtime at the next registry fetch cycle as specified by registryFetchIntervalSecondsr |
| eureka.client.on-demand-update-status-change | true | If set to true, local status updates via ApplicationInfoManager will trigger on-demand (but rate limited) register/updates to remote eureka servers |
| eureka.client.prefer-same-zone-eureka | true | Indicates whether or not this instance should try to use the eureka server in the same zone for latency and/or other reason. Ideally eureka clients are configured to talk to servers in the same zone The changes are effective at runtime at the next registry fetch cycle as specified by registryFetchIntervalSeconds |
| eureka.client.property-resolver |  |  |
| eureka.client.proxy-host |  | Gets the proxy host to eureka server if any. |
| eureka.client.proxy-password |  | Gets the proxy password if any. |
| eureka.client.proxy-port |  | Gets the proxy port to eureka server if any. |
| eureka.client.proxy-user-name |  | Gets the proxy user name if any. |
| eureka.client.region | us-east-1 | Gets the region (used in AWS datacenters) where this instance resides. |
| eureka.client.register-with-eureka | true | Indicates whether or not this instance should register its information with eureka server for discovery by others. In some cases, you do not want your instances to be discovered whereas you just want do discover other instances. |
| eureka.client.registry-fetch-interval-seconds | 30 | Indicates how often(in seconds) to fetch the registry information from the eureka server. |
| eureka.client.registry-refresh-single-vip-address |  | Indicates whether the client is only interested in the registry information for a single VIP. |
| eureka.client.service-url |  | Map of availability zone to list of fully qualified URLs to communicate with eureka server. Each value can be a single URL or a comma separated list of alternative locations. Typically the eureka server URLs carry protocol,host,port,context and version information if any. Example: <http://ec2-256-156-243-129.compute-1.amazonaws.com:7001/eureka/> The changes are effective at runtime at the next service url refresh cycle as specified by eurekaServiceUrlPollIntervalSeconds. |
| eureka.client.should-enforce-registration-at-init | false | Indicates whether the client should enforce registration during initialization. Defaults to false. |
| eureka.client.should-unregister-on-shutdown | true | Indicates whether the client should explicitly unregister itself from the remote server on client shutdown. |
| eureka.client.use-dns-for-fetching-service-urls | false | Indicates whether the eureka client should use the DNS mechanism to fetch a list of eureka servers to talk to. When the DNS name is updated to have additional servers, that information is used immediately after the eureka client polls for that information as specified in eurekaServiceUrlPollIntervalSeconds. Alternatively, the service urls can be returned serviceUrls, but the users should implement their own mechanism to return the updated list in case of changes. The changes are effective at runtime. |
| eureka.dashboard.enabled | true | Flag to enable the Eureka dashboard. Default true. |
| eureka.dashboard.path | / | The path to the Eureka dashboard (relative to the servlet path). Defaults to "/". |
| eureka.instance.a-s-g-name |  | Gets the AWS autoscaling group name associated with this instance. This information is specifically used in an AWS environment to automatically put an instance out of service after the instance is launched and it has been disabled for traffic.. |
| eureka.instance.app-group-name |  | Get the name of the application group to be registered with eureka. |
| eureka.instance.appname | unknown | Get the name of the application to be registered with eureka. |
| eureka.instance.data-center-info |  | Returns the data center this instance is deployed. This information is used to get some AWS specific instance information if the instance is deployed in AWS. |
| eureka.instance.default-address-resolution-order | [] |  |
| eureka.instance.environment |  |  |
| eureka.instance.health-check-url |  | Gets the absolute health check page URL for this instance. The users can provide the healthCheckUrlPath if the health check page resides in the same instance talking to eureka, else in the cases where the instance is a proxy for some other server, users can provide the full URL. If the full URL is provided it takes precedence. <p> It is normally used for making educated decisions based on the health of the instance - for example, it can be used to determine whether to proceed deployments to an entire farm or stop the deployments without causing further damage. The full URL should follow the format http://${eureka.hostname}:7001/ where the value ${eureka.hostname} is replaced at runtime. |
| eureka.instance.health-check-url-path |  | Gets the relative health check URL path for this instance. The health check page URL is then constructed out of the hostname and the type of communication - secure or unsecure as specified in securePort and nonSecurePort. It is normally used for making educated decisions based on the health of the instance - for example, it can be used to determine whether to proceed deployments to an entire farm or stop the deployments without causing further damage. |
| eureka.instance.home-page-url |  | Gets the absolute home page URL for this instance. The users can provide the homePageUrlPath if the home page resides in the same instance talking to eureka, else in the cases where the instance is a proxy for some other server, users can provide the full URL. If the full URL is provided it takes precedence. It is normally used for informational purposes for other services to use it as a landing page. The full URL should follow the format http://${eureka.hostname}:7001/ where the value ${eureka.hostname} is replaced at runtime. |
| eureka.instance.home-page-url-path | / | Gets the relative home page URL Path for this instance. The home page URL is then constructed out of the hostName and the type of communication - secure or unsecure. It is normally used for informational purposes for other services to use it as a landing page. |
| eureka.instance.hostname |  | The hostname if it can be determined at configuration time (otherwise it will be guessed from OS primitives). |
| eureka.instance.initial-status |  | Initial status to register with rmeote Eureka server. |
| eureka.instance.instance-enabled-onit | false | Indicates whether the instance should be enabled for taking traffic as soon as it is registered with eureka. Sometimes the application might need to do some pre-processing before it is ready to take traffic. |
| eureka.instance.instance-id |  | Get the unique Id (within the scope of the appName) of this instance to be registered with eureka. |
| eureka.instance.ip-address |  | Get the IPAdress of the instance. This information is for academic purposes only as the communication from other instances primarily happen using the information supplied in {@link #getHostName(boolean)}. |
| eureka.instance.lease-expiration-duration-in-seconds | 90 | Indicates the time in seconds that the eureka server waits since it received the last heartbeat before it can remove this instance from its view and there by disallowing traffic to this instance. Setting this value too long could mean that the traffic could be routed to the instance even though the instance is not alive. Setting this value too small could mean, the instance may be taken out of traffic because of temporary network glitches.This value to be set to atleast higher than the value specified in leaseRenewalIntervalInSeconds. |
| eureka.instance.lease-renewal-interval-in-seconds | 30 | Indicates how often (in seconds) the eureka client needs to send heartbeats to eureka server to indicate that it is still alive. If the heartbeats are not received for the period specified in leaseExpirationDurationInSeconds, eureka server will remove the instance from its view, there by disallowing traffic to this instance. Note that the instance could still not take traffic if it implements HealthCheckCallback and then decides to make itself unavailable. |
| eureka.instance.metadata-map |  | Gets the metadata name/value pairs associated with this instance. This information is sent to eureka server and can be used by other instances. |
| eureka.instance.namespace | eureka | Get the namespace used to find properties. Ignored in Spring Cloud. |
| eureka.instance.non-secure-port | 80 | Get the non-secure port on which the instance should receive traffic. |
| eureka.instance.non-secure-port-enabled | true | Indicates whether the non-secure port should be enabled for traffic or not. |
| eureka.instance.prefer-ip-address | false | Flag to say that, when guessing a hostname, the IP address of the server should be used in prference to the hostname reported by the OS. |
| eureka.instance.registry.default-open-for-traffic-count | 1 | Value used in determining when leases are cancelled, default to 1 for standalone. Should be set to 0 for peer replicated eurekas |
| eureka.instance.registry.expected-number-of-renews-per-min | 1 |  |
| eureka.instance.secure-health-check-url |  | Gets the absolute secure health check page URL for this instance. The users can provide the secureHealthCheckUrl if the health check page resides in the same instance talking to eureka, else in the cases where the instance is a proxy for some other server, users can provide the full URL. If the full URL is provided it takes precedence. <p> It is normally used for making educated decisions based on the health of the instance - for example, it can be used to determine whether to proceed deployments to an entire farm or stop the deployments without causing further damage. The full URL should follow the format http://${eureka.hostname}:7001/ where the value ${eureka.hostname} is replaced at runtime. |
| eureka.instance.secure-port | 443 | Get the Secure port on which the instance should receive traffic. |
| eureka.instance.secure-port-enabled | false | Indicates whether the secure port should be enabled for traffic or not. |
| eureka.instance.secure-virtual-host-name | unknown | Gets the secure virtual host name defined for this instance. This is typically the way other instance would find this instance by using the secure virtual host name.Think of this as similar to the fully qualified domain name, that the users of your services will need to find this instance. |
| eureka.instance.status-page-url |  | Gets the absolute status page URL path for this instance. The users can provide the statusPageUrlPath if the status page resides in the same instance talking to eureka, else in the cases where the instance is a proxy for some other server, users can provide the full URL. If the full URL is provided it takes precedence. It is normally used for informational purposes for other services to find about the status of this instance. Users can provide a simple HTML indicating what is the current status of the instance. |
| eureka.instance.status-page-url-path |  | Gets the relative status page URL path for this instance. The status page URL is then constructed out of the hostName and the type of communication - secure or unsecure as specified in securePort and nonSecurePort. It is normally used for informational purposes for other services to find about the status of this instance. Users can provide a simple HTML indicating what is the current status of the instance. |
| eureka.instance.virtual-host-name | unknown | Gets the virtual host name defined for this instance. This is typically the way other instance would find this instance by using the virtual host name.Think of this as similar to the fully qualified domain name, that the users of your services will need to find this instance. |
| eureka.server.a-s-g-cache-expiry-timeout-ms | 0 |  |
| eureka.server.a-s-g-query-timeout-ms | 300 |  |
| eureka.server.a-s-g-update-interval-ms | 0 |  |
| eureka.server.a-w-s-access-id |  |  |
| eureka.server.a-w-s-secret-key |  |  |
| eureka.server.batch-replication | false |  |
| eureka.server.binding-strategy |  |  |
| eureka.server.delta-retention-timer-interval-in-ms | 0 |  |
| eureka.server.disable-delta | false |  |
| eureka.server.disable-delta-for-remote-regions | false |  |
| eureka.server.disable-transparent-fallback-to-other-region | false |  |
| eureka.server.e-i-p-bind-rebind-retries | 3 |  |
| eureka.server.e-i-p-binding-retry-interval-ms | 0 |  |
| eureka.server.e-i-p-binding-retry-interval-ms-when-unbound | 0 |  |
| eureka.server.enable-replicated-request-compression | false |  |
| eureka.server.enable-self-preservation | true |  |
| eureka.server.eviction-interval-timer-in-ms | 0 |  |
| eureka.server.g-zip-content-from-remote-region | true |  |
| eureka.server.json-codec-name |  |  |
| eureka.server.list-auto-scaling-groups-role-name | ListAutoScalingGroups |  |
| eureka.server.log-identity-headers | true |  |
| eureka.server.max-elements-in-peer-replication-pool | 10000 |  |
| eureka.server.max-elements-in-status-replication-pool | 10000 |  |
| eureka.server.max-idle-thread-age-in-minutes-for-peer-replication | 15 |  |
| eureka.server.max-idle-thread-in-minutes-age-for-status-replication | 10 |  |
| eureka.server.max-threads-for-peer-replication | 20 |  |
| eureka.server.max-threads-for-status-replication | 1 |  |
| eureka.server.max-time-for-replication | 30000 |  |
| eureka.server.min-available-instances-for-peer-replication | -1 |  |
| eureka.server.min-threads-for-peer-replication | 5 |  |
| eureka.server.min-threads-for-status-replication | 1 |  |
| eureka.server.number-of-replication-retries | 5 |  |
| eureka.server.peer-eureka-nodes-update-interval-ms | 0 |  |
| eureka.server.peer-eureka-status-refresh-time-interval-ms | 0 |  |
| eureka.server.peer-node-connect-timeout-ms | 200 |  |
| eureka.server.peer-node-connection-idle-timeout-seconds | 30 |  |
| eureka.server.peer-node-read-timeout-ms | 200 |  |
| eureka.server.peer-node-total-connections | 1000 |  |
| eureka.server.peer-node-total-connections-per-host | 500 |  |
| eureka.server.prime-aws-replica-connections | true |  |
| eureka.server.property-resolver |  |  |
| eureka.server.rate-limiter-burst-size | 10 |  |
| eureka.server.rate-limiter-enabled | false |  |
| eureka.server.rate-limiter-full-fetch-average-rate | 100 |  |
| eureka.server.rate-limiter-privileged-clients |  |  |
| eureka.server.rate-limiter-registry-fetch-average-rate | 500 |  |
| eureka.server.rate-limiter-throttle-standard-clients | false |  |
| eureka.server.registry-sync-retries | 0 |  |
| eureka.server.registry-sync-retry-wait-ms | 0 |  |
| eureka.server.remote-region-app-whitelist |  |  |
| eureka.server.remote-region-connect-timeout-ms | 1000 |  |
| eureka.server.remote-region-connection-idle-timeout-seconds | 30 |  |
| eureka.server.remote-region-fetch-thread-pool-size | 20 |  |
| eureka.server.remote-region-read-timeout-ms | 1000 |  |
| eureka.server.remote-region-registry-fetch-interval | 30 |  |
| eureka.server.remote-region-total-connections | 1000 |  |
| eureka.server.remote-region-total-connections-per-host | 500 |  |
| eureka.server.remote-region-trust-store |  |  |
| eureka.server.remote-region-trust-store-password | changeit |  |
| eureka.server.remote-region-urls |  |  |
| eureka.server.remote-region-urls-with-name |  |  |
| eureka.server.renewal-percent-threshold | 0.85 |  |
| eureka.server.renewal-threshold-update-interval-ms | 0 |  |
| eureka.server.response-cache-auto-expiration-in-seconds | 180 |  |
| eureka.server.response-cache-update-interval-ms | 0 |  |
| eureka.server.retention-time-in-m-s-in-delta-queue | 0 |  |
| eureka.server.route53-bind-rebind-retries | 3 |  |
| eureka.server.route53-binding-retry-interval-ms | 0 |  |
| eureka.server.route53-domain-t-t-l | 30 |  |
| eureka.server.sync-when-timestamp-differs | true |  |
| eureka.server.use-read-only-response-cache | true |  |
| eureka.server.wait-time-in-ms-when-sync-empty | 0 |  |
| eureka.server.xml-codec-name |  |  |
| health.config.enabled | false | Flag to indicate that the config server health indicator should be installed. |
| health.config.time-to-live | 0 | Time to live for cached result, in milliseconds. Default 300000 (5 min). |
| hystrix.metrics.enabled | true | Enable Hystrix metrics polling. Defaults to true. |
| hystrix.metrics.polling-interval-ms | 2000 | Interval between subsequent polling of metrics. Defaults to 2000 ms. |
| management.endpoint.bindings.cache.time-to-live | 0ms | Maximum time that a response can be cached. |
| management.endpoint.bindings.enabled | true | Whether to enable the bindings endpoint. |
| management.endpoint.bus-env.enabled | true | Whether to enable the bus-env endpoint. |
| management.endpoint.bus-refresh.enabled | true | Whether to enable the bus-refresh endpoint. |
| management.endpoint.channels.cache.time-to-live | 0ms | Maximum time that a response can be cached. |
| management.endpoint.channels.enabled | true | Whether to enable the channels endpoint. |
| management.endpoint.consul.cache.time-to-live | 0ms | Maximum time that a response can be cached. |
| management.endpoint.consul.enabled | true | Whether to enable the consul endpoint. |
| management.endpoint.env.post.enabled | true | Enable changing the Environment through a POST to /env. |
| management.endpoint.features.cache.time-to-live | 0ms | Maximum time that a response can be cached. |
| management.endpoint.features.enabled | true | Whether to enable the features endpoint. |
| management.endpoint.gateway.enabled | true | Whether to enable the gateway endpoint. |
| management.endpoint.hystrix.config |  | Hystrix settings. These are traditionally set using servlet parameters. Refer to the documentation of Hystrix for more details. |
| management.endpoint.hystrix.stream.enabled | true | Whether to enable the hystrix.stream endpoint. |
| management.endpoint.pause.enabled | true | Enable the /pause endpoint (to send Lifecycle.stop()). |
| management.endpoint.refresh.enabled | true | Enable the /refresh endpoint to refresh configuration and re-initialize refresh scoped beans. |
| management.endpoint.restart.enabled | true | Enable the /restart endpoint to restart the application context. |
| management.endpoint.resume.enabled | true | Enable the /resume endpoint (to send Lifecycle.start()). |
| management.endpoint.service-registry.cache.time-to-live | 0ms | Maximum time that a response can be cached. |
| management.endpoint.service-registry.enabled | true | Whether to enable the service-registry endpoint. |
| management.health.refresh.enabled | true | Enable the health endpoint for the refresh scope. |
| management.health.zookeeper.enabled | true | Enable the health endpoint for zookeeper. |
| proxy.auth.load-balanced | false |  |
| proxy.auth.routes |  | Authentication strategy per route. |
| ribbon.eager-load.clients |  |  |
| ribbon.eager-load.enabled | false |  |
| ribbon.secure-ports |  |  |
| spring.cloud.bus.ack.destination-service |  | Service that wants to listen to acks. By default null (meaning all services). |
| spring.cloud.bus.ack.enabled | true | Flag to switch off acks (default on). |
| spring.cloud.bus.destination | springCloudBus | Name of Spring Cloud Stream destination for messages. |
| spring.cloud.bus.enabled | true | Flag to indicate that the bus is enabled. |
| spring.cloud.bus.env.enabled | true | Flag to switch off environment change events (default on). |
| spring.cloud.bus.id | application | The identifier for this application instance. |
| spring.cloud.bus.refresh.enabled | true | Flag to switch off refresh events (default on). |
| spring.cloud.bus.trace.enabled | false | Flag to switch on tracing of acks (default off). |
| spring.cloud.cloudfoundry.discovery.default-server-port | 80 | Port to use when no port is defined by ribbon. |
| spring.cloud.cloudfoundry.discovery.enabled | true | Flag to indicate that discovery is enabled. |
| spring.cloud.cloudfoundry.discovery.heartbeat-frequency | 5000 | Frequency in milliseconds of poll for heart beat. The client will poll on this frequency and broadcast a list of service ids. |
| spring.cloud.cloudfoundry.org |  | Organization name to initially target. |
| spring.cloud.cloudfoundry.password |  | Password for user to authenticate and obtain token. |
| spring.cloud.cloudfoundry.skip-ssl-validation | false |  |
| spring.cloud.cloudfoundry.space |  | Space name to initially target. |
| spring.cloud.cloudfoundry.url |  | URL of Cloud Foundry API (Cloud Controller). |
| spring.cloud.cloudfoundry.username |  | Username to authenticate (usually an email address). |
| spring.cloud.config.allow-override | true | Flag to indicate that {@link #isOverrideSystemProperties() systemPropertiesOverride} can be used. Set to false to prevent users from changing the default accidentally. Default true. |
| spring.cloud.config.discovery.enabled | false | Flag to indicate that config server discovery is enabled (config server URL will be looked up via discovery). |
| spring.cloud.config.discovery.service-id | configserver | Service id to locate config server. |
| spring.cloud.config.enabled | true | Flag to say that remote configuration is enabled. Default true; |
| spring.cloud.config.fail-fast | false | Flag to indicate that failure to connect to the server is fatal (default false). |
| spring.cloud.config.headers |  | Additional headers used to create the client request. |
| spring.cloud.config.label |  | The label name to use to pull remote configuration properties. The default is set on the server (generally "master" for a git based server). |
| spring.cloud.config.name |  | Name of application used to fetch remote properties. |
| spring.cloud.config.override-none | false | Flag to indicate that when {@link #setAllowOverride(boolean) allowOverride} is true, external properties should take lowest priority, and not override any existing property sources (including local config files). Default false. |
| spring.cloud.config.override-system-properties | true | Flag to indicate that the external properties should override system properties. Default true. |
| spring.cloud.config.password |  | The password to use (HTTP Basic) when contacting the remote server. |
| spring.cloud.config.profile | default | The default profile to use when fetching remote configuration (comma-separated). Default is "default". |
| spring.cloud.config.request-read-timeout | 0 | timeout on waiting to read data from the Config Server. |
| spring.cloud.config.retry.initial-interval | 1000 | Initial retry interval in milliseconds. |
| spring.cloud.config.retry.max-attempts | 6 | Maximum number of attempts. |
| spring.cloud.config.retry.max-interval | 2000 | Maximum interval for backoff. |
| spring.cloud.config.retry.multiplier | 1.1 | Multiplier for next interval. |
| spring.cloud.config.send-state | true | Flag to indicate whether to send state. Default true. |
| spring.cloud.config.server.accept-empty | true | Flag to indicate that If HTTP 404 needs to be sent if Application is not Found |
| spring.cloud.config.server.bootstrap | false | Flag indicating that the config server should initialize its own Environment with properties from the remote repository. Off by default because it delays startup but can be useful when embedding the server in another application. |
| spring.cloud.config.server.default-application-name | application | Default application name when incoming requests do not have a specific one. |
| spring.cloud.config.server.default-label |  | Default repository label when incoming requests do not have a specific label. |
| spring.cloud.config.server.default-profile | default | Default application profile when incoming requests do not have a specific one. |
| spring.cloud.config.server.encrypt.enabled | true | Enable decryption of environment properties before sending to client. |
| spring.cloud.config.server.git.basedir |  | Base directory for local working copy of repository. |
| spring.cloud.config.server.git.clone-on-start | false | Flag to indicate that the repository should be cloned on startup (not on demand). Generally leads to slower startup but faster first query. |
| spring.cloud.config.server.git.default-label |  | The default label to be used with the remore repository |
| spring.cloud.config.server.git.delete-untracked-branches | false | Flag to indicate that the branch should be deleted locally if it’s origin tracked branch was removed. |
| spring.cloud.config.server.git.force-pull | false | Flag to indicate that the repository should force pull. If true discard any local changes and take from remote repository. |
| spring.cloud.config.server.git.host-key |  | Valid SSH host key. Must be set if hostKeyAlgorithm is also set. |
| spring.cloud.config.server.git.host-key-algorithm |  | One of ssh-dss, ssh-rsa, ecdsa-sha2-nistp256, ecdsa-sha2-nistp384, or ecdsa-sha2-nistp521. Must be set if hostKey is also set. |
| spring.cloud.config.server.git.ignore-local-ssh-settings | false | If true, use property-based instead of file-based SSH config. |
| spring.cloud.config.server.git.known-hosts-file |  | Location of custom .known\_hosts file. |
| spring.cloud.config.server.git.order |  | The order of the environment repository. |
| spring.cloud.config.server.git.passphrase |  | Passphrase for unlocking your ssh private key. |
| spring.cloud.config.server.git.password |  | Password for authentication with remote repository. |
| spring.cloud.config.server.git.preferred-authentications |  | Override server authentication method order. This should allow for evading login prompts if server has keyboard-interactive authentication before the publickey method. |
| spring.cloud.config.server.git.private-key |  | Valid SSH private key. Must be set if ignoreLocalSshSettings is true and Git URI is SSH format. |
| spring.cloud.config.server.git.proxy |  | HTTP proxy configuration. |
| spring.cloud.config.server.git.refresh-rate | 0 | Time (in seconds) between refresh of the git repository |
| spring.cloud.config.server.git.repos |  | Map of repository identifier to location and other properties. |
| spring.cloud.config.server.git.search-paths |  | Search paths to use within local working copy. By default searches only the root. |
| spring.cloud.config.server.git.skip-ssl-validation | false | Flag to indicate that SSL certificate validation should be bypassed when communicating with a repository served over an HTTPS connection. |
| spring.cloud.config.server.git.strict-host-key-checking | true | If false, ignore errors with host key |
| spring.cloud.config.server.git.timeout | 5 | Timeout (in seconds) for obtaining HTTP or SSH connection (if applicable), defaults to 5 seconds. |
| spring.cloud.config.server.git.uri |  | URI of remote repository. |
| spring.cloud.config.server.git.username |  | Username for authentication with remote repository. |
| spring.cloud.config.server.health.repositories |  |  |
| spring.cloud.config.server.jdbc.order | 0 |  |
| spring.cloud.config.server.jdbc.sql | SELECT KEY, VALUE from PROPERTIES where APPLICATION=? and PROFILE=? and LABEL=? | SQL used to query database for keys and values |
| spring.cloud.config.server.native.add-label-locations | true | Flag to determine whether label locations should be added. |
| spring.cloud.config.server.native.default-label | master |  |
| spring.cloud.config.server.native.fail-on-error | false | Flag to determine how to handle exceptions during decryption (default false). |
| spring.cloud.config.server.native.order |  |  |
| spring.cloud.config.server.native.search-locations | [] | Locations to search for configuration files. Defaults to the same as a Spring Boot app so [classpath:/,classpath:/config/,file:./,file:./config/]. |
| spring.cloud.config.server.native.version |  | Version string to be reported for native repository |
| spring.cloud.config.server.overrides |  | Extra map for a property source to be sent to all clients unconditionally. |
| spring.cloud.config.server.prefix |  | Prefix for configuration resource paths (default is empty). Useful when embedding in another application when you don’t want to change the context path or servlet path. |
| spring.cloud.config.server.strip-document-from-yaml | true | Flag to indicate that YAML documents that are text or collections (not a map) should be returned in "native" form. |
| spring.cloud.config.server.svn.basedir |  | Base directory for local working copy of repository. |
| spring.cloud.config.server.svn.default-label |  | The default label to be used with the remore repository |
| spring.cloud.config.server.svn.order |  | The order of the environment repository. |
| spring.cloud.config.server.svn.passphrase |  | Passphrase for unlocking your ssh private key. |
| spring.cloud.config.server.svn.password |  | Password for authentication with remote repository. |
| spring.cloud.config.server.svn.search-paths |  | Search paths to use within local working copy. By default searches only the root. |
| spring.cloud.config.server.svn.strict-host-key-checking | true | Reject incoming SSH host keys from remote servers not in the known host list. |
| spring.cloud.config.server.svn.uri |  | URI of remote repository. |
| spring.cloud.config.server.svn.username |  | Username for authentication with remote repository. |
| spring.cloud.config.server.vault.backend | secret | Vault backend. Defaults to secret. |
| spring.cloud.config.server.vault.default-key | application | The key in vault shared by all applications. Defaults to application. Set to empty to disable. |
| spring.cloud.config.server.vault.host | 127.0.0.1 | Vault host. Defaults to 127.0.0.1. |
| spring.cloud.config.server.vault.kv-version | 1 | Value to indicate which version of Vault kv backend is used. Defaults to 1. |
| spring.cloud.config.server.vault.order |  |  |
| spring.cloud.config.server.vault.port | 8200 | Vault port. Defaults to 8200. |
| spring.cloud.config.server.vault.profile-separator | , | Vault profile separator. Defaults to comma. |
| spring.cloud.config.server.vault.proxy |  | HTTP proxy configuration. |
| spring.cloud.config.server.vault.scheme | http | Vault scheme. Defaults to http. |
| spring.cloud.config.server.vault.skip-ssl-validation | false | Flag to indicate that SSL certificate validation should be bypassed when communicating with a repository served over an HTTPS connection. |
| spring.cloud.config.server.vault.timeout | 5 | Timeout (in seconds) for obtaining HTTP connection, defaults to 5 seconds. |
| spring.cloud.config.token |  | Security Token passed thru to underlying environment repository. |
| spring.cloud.config.uri | [[http://localhost:8888](http://localhost:8888/)] | The URI of the remote server (default [http://localhost:8888](http://localhost:8888/)). |
| spring.cloud.config.username |  | The username to use (HTTP Basic) when contacting the remote server. |
| spring.cloud.consul.config.acl-token |  |  |
| spring.cloud.consul.config.data-key | data | If format is Format.PROPERTIES or Format.YAML then the following field is used as key to look up consul for configuration. |
| spring.cloud.consul.config.default-context | application |  |
| spring.cloud.consul.config.enabled | true |  |
| spring.cloud.consul.config.fail-fast | true | Throw exceptions during config lookup if true, otherwise, log warnings. |
| spring.cloud.consul.config.format |  |  |
| spring.cloud.consul.config.name |  | Alternative to spring.application.name to use in looking up values in consul KV. |
| spring.cloud.consul.config.prefix | config |  |
| spring.cloud.consul.config.profile-separator | , |  |
| spring.cloud.consul.config.watch.delay | 1000 | The value of the fixed delay for the watch in millis. Defaults to 1000. |
| spring.cloud.consul.config.watch.enabled | true | If the watch is enabled. Defaults to true. |
| spring.cloud.consul.config.watch.wait-time | 55 | The number of seconds to wait (or block) for watch query, defaults to 55. Needs to be less than default ConsulClient (defaults to 60). To increase ConsulClient timeout create a ConsulClient bean with a custom ConsulRawClient with a custom HttpClient. |
| spring.cloud.consul.discovery.acl-token |  |  |
| spring.cloud.consul.discovery.catalog-services-watch-delay | 1000 | The delay between calls to watch consul catalog in millis, default is 1000. |
| spring.cloud.consul.discovery.catalog-services-watch-timeout | 2 | The number of seconds to block while watching consul catalog, default is 2. |
| spring.cloud.consul.discovery.datacenters |  | Map of serviceId’s → datacenter to query for in server list. This allows looking up services in another datacenters. |
| spring.cloud.consul.discovery.default-query-tag |  | Tag to query for in service list if one is not listed in serverListQueryTags. |
| spring.cloud.consul.discovery.default-zone-metadata-name | zone | Service instance zone comes from metadata. This allows changing the metadata tag name. |
| spring.cloud.consul.discovery.deregister | true | Disable automatic de-registration of service in consul. |
| spring.cloud.consul.discovery.enabled | true | Is service discovery enabled? |
| spring.cloud.consul.discovery.fail-fast | true | Throw exceptions during service registration if true, otherwise, log warnings (defaults to true). |
| spring.cloud.consul.discovery.health-check-critical-timeout |  | Timeout to deregister services critical for longer than timeout (e.g. 30m). Requires consul version 7.x or higher. |
| spring.cloud.consul.discovery.health-check-interval | 10s | How often to perform the health check (e.g. 10s), defaults to 10s. |
| spring.cloud.consul.discovery.health-check-path | /actuator/health | Alternate server path to invoke for health checking |
| spring.cloud.consul.discovery.health-check-timeout |  | Timeout for health check (e.g. 10s). |
| spring.cloud.consul.discovery.health-check-tls-skip-verify |  | Skips certificate verification during service checks if true, otherwise runs certificate verification. |
| spring.cloud.consul.discovery.health-check-url |  | Custom health check url to override default |
| spring.cloud.consul.discovery.heartbeat.enabled | false |  |
| spring.cloud.consul.discovery.heartbeat.interval-ratio |  |  |
| spring.cloud.consul.discovery.heartbeat.ttl-unit | s |  |
| spring.cloud.consul.discovery.heartbeat.ttl-value | 30 |  |
| spring.cloud.consul.discovery.hostname |  | Hostname to use when accessing server |
| spring.cloud.consul.discovery.instance-group |  | Service instance group |
| spring.cloud.consul.discovery.instance-id |  | Unique service instance id |
| spring.cloud.consul.discovery.instance-zone |  | Service instance zone |
| spring.cloud.consul.discovery.ip-address |  | IP address to use when accessing service (must also set preferIpAddress to use) |
| spring.cloud.consul.discovery.lifecycle.enabled | true |  |
| spring.cloud.consul.discovery.management-port |  | Port to register the management service under (defaults to management port) |
| spring.cloud.consul.discovery.management-suffix | management | Suffix to use when registering management service |
| spring.cloud.consul.discovery.management-tags |  | Tags to use when registering management service |
| spring.cloud.consul.discovery.port |  | Port to register the service under (defaults to listening port) |
| spring.cloud.consul.discovery.prefer-agent-address | false | Source of how we will determine the address to use |
| spring.cloud.consul.discovery.prefer-ip-address | false | Use ip address rather than hostname during registration |
| spring.cloud.consul.discovery.query-passing | false | Add the 'passing` parameter to /v1/health/service/serviceName. This pushes health check passing to the server. |
| spring.cloud.consul.discovery.register | true | Register as a service in consul. |
| spring.cloud.consul.discovery.register-health-check | true | Register health check in consul. Useful during development of a service. |
| spring.cloud.consul.discovery.scheme | http | Whether to register an http or https service |
| spring.cloud.consul.discovery.server-list-query-tags |  | Map of serviceId’s → tag to query for in server list. This allows filtering services by a single tag. |
| spring.cloud.consul.discovery.service-name |  | Service name |
| spring.cloud.consul.discovery.tags |  | Tags to use when registering service |
| spring.cloud.consul.enabled | true | Is spring cloud consul enabled |
| spring.cloud.consul.host | localhost | Consul agent hostname. Defaults to 'localhost'. |
| spring.cloud.consul.port | 8500 | Consul agent port. Defaults to '8500'. |
| spring.cloud.consul.retry.initial-interval | 1000 | Initial retry interval in milliseconds. |
| spring.cloud.consul.retry.max-attempts | 6 | Maximum number of attempts. |
| spring.cloud.consul.retry.max-interval | 2000 | Maximum interval for backoff. |
| spring.cloud.consul.retry.multiplier | 1.1 | Multiplier for next interval. |
| spring.cloud.consul.scheme |  | Consul agent scheme (HTTP/HTTPS). If there is no scheme in address - client will use HTTP. |
| spring.cloud.consul.tls.certificate-password |  | Password to open the certificate. |
| spring.cloud.consul.tls.certificate-path |  | File path to the certificate. |
| spring.cloud.consul.tls.key-store-instance-type |  | Type of key framework to use. |
| spring.cloud.consul.tls.key-store-password |  | Password to an external keystore |
| spring.cloud.consul.tls.key-store-path |  | Path to an external keystore |
| spring.cloud.discovery.client.health-indicator.enabled | true |  |
| spring.cloud.discovery.client.health-indicator.include-description | false |  |
| spring.cloud.discovery.client.simple.instances |  |  |
| spring.cloud.discovery.client.simple.local.metadata |  | Metadata for the service instance. Can be used by discovery clients to modify their behaviour per instance, e.g. when load balancing. |
| spring.cloud.discovery.client.simple.local.service-id |  | The identifier or name for the service. Multiple instances might share the same service id. |
| spring.cloud.discovery.client.simple.local.uri |  | The URI of the service instance. Will be parsed to extract the scheme, hos and port. |
| spring.cloud.gateway.default-filters |  | List of filter definitions that are applied to every route. |
| spring.cloud.gateway.discovery.locator.enabled | false | Flag that enables DiscoveryClient gateway integration |
| spring.cloud.gateway.discovery.locator.filters |  |  |
| spring.cloud.gateway.discovery.locator.include-expression | true | SpEL expression that will evaluate whether to include a service in gateway integration or not, defaults to: true |
| spring.cloud.gateway.discovery.locator.lower-case-service-id | false | Option to lower case serviceId in predicates and filters, defaults to false. Useful with eureka when it automatically uppercases serviceId. so MYSERIVCE, would match /myservice/\*\* |
| spring.cloud.gateway.discovery.locator.predicates |  |  |
| spring.cloud.gateway.discovery.locator.route-id-prefix |  | The prefix for the routeId, defaults to discoveryClient.getClass().getSimpleName() + "\_". Service Id will be appended to create the routeId. |
| spring.cloud.gateway.discovery.locator.url-expression | 'lb://'+serviceId | SpEL expression that create the uri for each route, defaults to: 'lb://'+serviceId |
| spring.cloud.gateway.filter.remove-hop-by-hop.headers |  |  |
| spring.cloud.gateway.filter.remove-hop-by-hop.order |  |  |
| spring.cloud.gateway.filter.secure-headers.content-security-policy | default-src 'self' https:; font-src 'self' https: data:; img-src 'self' https: data:; object-src 'none'; script-src https:; style-src 'self' https: 'unsafe-inline' |  |
| spring.cloud.gateway.filter.secure-headers.content-type-options | nosniff |  |
| spring.cloud.gateway.filter.secure-headers.download-options | noopen |  |
| spring.cloud.gateway.filter.secure-headers.frame-options | DENY |  |
| spring.cloud.gateway.filter.secure-headers.permitted-cross-domain-policies | none |  |
| spring.cloud.gateway.filter.secure-headers.referrer-policy | no-referrer |  |
| spring.cloud.gateway.filter.secure-headers.strict-transport-security | max-age=631138519 |  |
| spring.cloud.gateway.filter.secure-headers.xss-protection-header | 1 ; mode=block |  |
| spring.cloud.gateway.globalcors.cors-configurations |  |  |
| spring.cloud.gateway.httpclient.connect-timeout |  | The connect timeout in millis, the default is 45s. |
| spring.cloud.gateway.httpclient.pool.acquire-timeout |  | Only for type FIXED, the maximum time in millis to wait for aquiring. |
| spring.cloud.gateway.httpclient.pool.max-connections |  | Only for type FIXED, the maximum number of connections before starting pending acquisition on existing ones. |
| spring.cloud.gateway.httpclient.pool.name | proxy | The channel pool map name, defaults to proxy. |
| spring.cloud.gateway.httpclient.pool.type |  | Type of pool for HttpClient to use, defaults to ELASTIC. |
| spring.cloud.gateway.httpclient.proxy.host |  | Hostname for proxy configuration of Netty HttpClient. |
| spring.cloud.gateway.httpclient.proxy.non-proxy-hosts-pattern |  | Regular expression (Java) for a configured list of hosts that should be reached directly, bypassing the proxy |
| spring.cloud.gateway.httpclient.proxy.password |  | Password for proxy configuration of Netty HttpClient. |
| spring.cloud.gateway.httpclient.proxy.port |  | Port for proxy configuration of Netty HttpClient. |
| spring.cloud.gateway.httpclient.proxy.username |  | Username for proxy configuration of Netty HttpClient. |
| spring.cloud.gateway.httpclient.response-timeout |  | The response timeout. |
| spring.cloud.gateway.httpclient.ssl.trusted-x509-certificates |  |  |
| spring.cloud.gateway.httpclient.ssl.use-insecure-trust-manager | false | Installs the netty InsecureTrustManagerFactory. This is insecure and not suitable for production. |
| spring.cloud.gateway.proxy.headers |  | Fixed header values that will be added to all downstream requests. |
| spring.cloud.gateway.proxy.sensitive |  | A set of sensitive header names that will not be sent downstream by default. |
| spring.cloud.gateway.redis-rate-limiter.burst-capacity-header | X-RateLimit-Burst-Capacity | The name of the header that returns the burst capacity configuration. |
| spring.cloud.gateway.redis-rate-limiter.config |  |  |
| spring.cloud.gateway.redis-rate-limiter.include-headers | true | Whether or not to include headers containing rate limiter information, defaults to true. |
| spring.cloud.gateway.redis-rate-limiter.remaining-header | X-RateLimit-Remaining | The name of the header that returns number of remaining requests during the current second. |
| spring.cloud.gateway.redis-rate-limiter.replenish-rate-header | X-RateLimit-Replenish-Rate | The name of the header that returns the replenish rate configuration. |
| spring.cloud.gateway.routes |  | List of Routes |
| spring.cloud.gateway.streaming-media-types |  |  |
| spring.cloud.gateway.x-forwarded.enabled | true | If the XForwardedHeadersFilter is enabled. |
| spring.cloud.gateway.x-forwarded.for-append | true | If appending X-Forwarded-For as a list is enabled. |
| spring.cloud.gateway.x-forwarded.for-enabled | true | If X-Forwarded-For is enabled. |
| spring.cloud.gateway.x-forwarded.host-append | true | If appending X-Forwarded-Host as a list is enabled. |
| spring.cloud.gateway.x-forwarded.host-enabled | true | If X-Forwarded-Host is enabled. |
| spring.cloud.gateway.x-forwarded.order | 0 | The order of the XForwardedHeadersFilter. |
| spring.cloud.gateway.x-forwarded.port-append | true | If appending X-Forwarded-Port as a list is enabled. |
| spring.cloud.gateway.x-forwarded.port-enabled | true | If X-Forwarded-Port is enabled. |
| spring.cloud.gateway.x-forwarded.proto-append | true | If appending X-Forwarded-Proto as a list is enabled. |
| spring.cloud.gateway.x-forwarded.proto-enabled | true | If X-Forwarded-Proto is enabled. |
| spring.cloud.hypermedia.refresh.fixed-delay | 5000 |  |
| spring.cloud.hypermedia.refresh.initial-delay | 10000 |  |
| spring.cloud.inetutils.default-hostname | localhost | The default hostname. Used in case of errors. |
| spring.cloud.inetutils.default-ip-address | 127.0.0.1 | The default ipaddress. Used in case of errors. |
| spring.cloud.inetutils.ignored-interfaces |  | List of Java regex expressions for network interfaces that will be ignored. |
| spring.cloud.inetutils.preferred-networks |  | List of Java regex expressions for network addresses that will be preferred. |
| spring.cloud.inetutils.timeout-seconds | 1 | Timeout in seconds for calculating hostname. |
| spring.cloud.inetutils.use-only-site-local-interfaces | false | Use only interfaces with site local addresses. See {@link InetAddress#isSiteLocalAddress()} for more details. |
| spring.cloud.loadbalancer.retry.enabled | true |  |
| spring.cloud.refresh.extra-refreshable | true | Additional class names for beans to post process into refresh scope. |
| spring.cloud.service-registry.auto-registration.enabled | true | If Auto-Service Registration is enabled, default to true. |
| spring.cloud.service-registry.auto-registration.fail-fast | false | Should startup fail if there is no AutoServiceRegistration, default to false. |
| spring.cloud.service-registry.auto-registration.register-management | true | Whether to register the management as a service, defaults to true |
| spring.cloud.stream.binders |  | Additional per-binder properties (see {@link BinderProperties}) if more then one binder of the same type is used (i.e., connect to multiple instances of RabbitMq). Here you can specify multiple binder configurations, each with different environment settings. For example; spring.cloud.stream.binders.rabbit1.environment. . . , spring.cloud.stream.binders.rabbit2.environment. . . |
| spring.cloud.stream.binding-retry-interval | 30 | Retry interval (in seconds) used to schedule binding attempts. Default: 30 sec. |
| spring.cloud.stream.bindings |  | Additional binding properties (see {@link BinderProperties}) per binding name (e.g., 'input`).  For example; This sets the content-type for the 'input' binding of a Sink application: 'spring.cloud.stream.bindings.input.contentType=text/plain' |
| spring.cloud.stream.consul.binder.event-timeout | 5 |  |
| spring.cloud.stream.default-binder |  | The name of the binder to use by all bindings in the event multiple binders available (e.g., 'rabbit'); |
| spring.cloud.stream.dynamic-destinations | [] | A list of destinations that can be bound dynamically. If set, only listed destinations can be bound. |
| spring.cloud.stream.instance-count | 1 | The number of deployed instances of an application. Default: 1. NOTE: Could also be managed per individual binding "spring.cloud.stream.bindings.foo.consumer.instance-count" where 'foo' is the name of the binding. |
| spring.cloud.stream.instance-index | 0 | The instance id of the application: a number from 0 to instanceCount-1. Used for partitioning and with Kafka. NOTE: Could also be managed per individual binding "spring.cloud.stream.bindings.foo.consumer.instance-index" where 'foo' is the name of the binding. |
| spring.cloud.stream.integration.message-handler-not-propagated-headers |  | Message header names that will NOT be copied from the inbound message. |
| spring.cloud.stream.metrics.export-properties |  | List of properties that are going to be appended to each message. This gets populate by onApplicationEvent, once the context refreshes to avoid overhead of doing per message basis. |
| spring.cloud.stream.metrics.key |  | The name of the metric being emitted. Should be an unique value per application. Defaults to: ${spring.application.name:${vcap.application.name:${spring.config.name:application}}} |
| spring.cloud.stream.metrics.meter-filter |  | Pattern to control the 'meters' one wants to capture. By default all 'meters' will be captured. For example, 'spring.integration.\*' will only capture metric information for meters whose name starts with 'spring.integration'. |
| spring.cloud.stream.metrics.properties |  | Application properties that should be added to the metrics payload For example: spring.application\*\* |
| spring.cloud.stream.metrics.schedule-interval | 60s | Interval expressed as Duration for scheduling metrics snapshots publishing. Defaults to 60 seconds |
| spring.cloud.stream.rabbit.binder.admin-addresses | [] | Urls for management plugins; only needed for queue affinity. |
| spring.cloud.stream.rabbit.binder.admin-adresses |  |  |
| spring.cloud.stream.rabbit.binder.compression-level | 0 | Compression level for compressed bindings; see 'java.util.zip.Deflator'. |
| spring.cloud.stream.rabbit.binder.connection-name-prefix |  | Prefix for connection names from this binder. |
| spring.cloud.stream.rabbit.binder.nodes | [] | Cluster member node names; only needed for queue affinity. |
| spring.cloud.stream.rabbit.bindings |  |  |
| spring.cloud.vault.app-id.app-id-path | app-id | Mount path of the AppId authentication backend. |
| spring.cloud.vault.app-id.network-interface |  | Network interface hint for the "MAC\_ADDRESS" UserId mechanism. |
| spring.cloud.vault.app-id.user-id | MAC\_ADDRESS | UserId mechanism. Can be either "MAC\_ADDRESS", "IP\_ADDRESS", a string or a class name. |
| spring.cloud.vault.app-role.app-role-path | approle | Mount path of the AppRole authentication backend. |
| spring.cloud.vault.app-role.role |  | Name of the role, optional, used for pull-mode. |
| spring.cloud.vault.app-role.role-id |  | The RoleId. |
| spring.cloud.vault.app-role.secret-id |  | The SecretId. |
| spring.cloud.vault.application-name | application | Application name for AppId authentication. |
| spring.cloud.vault.authentication |  |  |
| spring.cloud.vault.aws-ec2.aws-ec2-path | aws-ec2 | Mount path of the AWS-EC2 authentication backend. |
| spring.cloud.vault.aws-ec2.identity-document | <http://169.254.169.254/latest/dynamic/instance-identity/pkcs7> | URL of the AWS-EC2 PKCS7 identity document. |
| spring.cloud.vault.aws-ec2.nonce |  | Nonce used for AWS-EC2 authentication. An empty nonce defaults to nonce generation. |
| spring.cloud.vault.aws-ec2.role |  | Name of the role, optional. |
| spring.cloud.vault.aws-iam.aws-path | aws | Mount path of the AWS authentication backend. |
| spring.cloud.vault.aws-iam.role |  | Name of the role, optional. Defaults to the friendly IAM name if not set. |
| spring.cloud.vault.aws-iam.server-name |  | Name of the server used to set {@code X-Vault-AWS-IAM-Server-ID} header in the headers of login requests. |
| spring.cloud.vault.aws.access-key-property | cloud.aws.credentials.accessKey | Target property for the obtained access key. |
| spring.cloud.vault.aws.backend | aws | aws backend path. |
| spring.cloud.vault.aws.enabled | false | Enable aws backend usage. |
| spring.cloud.vault.aws.role |  | Role name for credentials. |
| spring.cloud.vault.aws.secret-key-property | cloud.aws.credentials.secretKey | Target property for the obtained secret key. |
| spring.cloud.vault.cassandra.backend | cassandra | Cassandra backend path. |
| spring.cloud.vault.cassandra.enabled | false | Enable cassandra backend usage. |
| spring.cloud.vault.cassandra.password-property | spring.data.cassandra.password | Target property for the obtained password. |
| spring.cloud.vault.cassandra.role |  | Role name for credentials. |
| spring.cloud.vault.cassandra.username-property | spring.data.cassandra.username | Target property for the obtained username. |
| spring.cloud.vault.config.lifecycle.enabled | true | Enable lifecycle management. |
| spring.cloud.vault.config.order | 0 | Used to set a {@link org.springframework.core.env.PropertySource} priority. This is useful to use Vault as an override on other property sources. @see org.springframework.core.PriorityOrdered |
| spring.cloud.vault.connection-timeout | 5000 | Connection timeout; |
| spring.cloud.vault.consul.backend | consul | Consul backend path. |
| spring.cloud.vault.consul.enabled | false | Enable consul backend usage. |
| spring.cloud.vault.consul.role |  | Role name for credentials. |
| spring.cloud.vault.consul.token-property | spring.cloud.consul.token | Target property for the obtained token. |
| spring.cloud.vault.database.backend | database | Database backend path. |
| spring.cloud.vault.database.enabled | false | Enable database backend usage. |
| spring.cloud.vault.database.password-property | spring.datasource.password | Target property for the obtained password. |
| spring.cloud.vault.database.role |  | Role name for credentials. |
| spring.cloud.vault.database.username-property | spring.datasource.username | Target property for the obtained username. |
| spring.cloud.vault.discovery.enabled | false | Flag to indicate that Vault server discovery is enabled (vault server URL will be looked up via discovery). |
| spring.cloud.vault.discovery.service-id | vault | Service id to locate Vault. |
| spring.cloud.vault.enabled | true | Enable Vault config server. |
| spring.cloud.vault.fail-fast | false | Fail fast if data cannot be obtained from Vault. |
| spring.cloud.vault.generic.application-name | application | Application name to be used for the context. |
| spring.cloud.vault.generic.backend | secret | Name of the default backend. |
| spring.cloud.vault.generic.default-context | application | Name of the default context. |
| spring.cloud.vault.generic.enabled | true | Enable the generic backend. |
| spring.cloud.vault.generic.profile-separator | / | Profile-separator to combine application name and profile. |
| spring.cloud.vault.host | localhost | Vault server host. |
| spring.cloud.vault.kubernetes.kubernetes-path | kubernetes | Mount path of the Kubernetes authentication backend. |
| spring.cloud.vault.kubernetes.role |  | Name of the role against which the login is being attempted. |
| spring.cloud.vault.kubernetes.service-account-token-file | /var/run/secrets/kubernetes.io/serviceaccount/token | Path to the service account token file. |
| spring.cloud.vault.kv.application-name | application | Application name to be used for the context. |
| spring.cloud.vault.kv.backend | secret | Name of the default backend. |
| spring.cloud.vault.kv.backend-version | 2 | Key-Value backend version. Currently supported versions are: <ul> <li>Version 1 (unversioned key-value backend).</li> <li>Version 2 (versioned key-value backend).</li> </ul> |
| spring.cloud.vault.kv.default-context | application | Name of the default context. |
| spring.cloud.vault.kv.enabled | false | Enable the kev-value backend. |
| spring.cloud.vault.kv.profile-separator | / | Profile-separator to combine application name and profile. |
| spring.cloud.vault.mongodb.backend | mongodb | Cassandra backend path. |
| spring.cloud.vault.mongodb.enabled | false | Enable mongodb backend usage. |
| spring.cloud.vault.mongodb.password-property | spring.data.mongodb.password | Target property for the obtained password. |
| spring.cloud.vault.mongodb.role |  | Role name for credentials. |
| spring.cloud.vault.mongodb.username-property | spring.data.mongodb.username | Target property for the obtained username. |
| spring.cloud.vault.mysql.backend | mysql | mysql backend path. |
| spring.cloud.vault.mysql.enabled | false | Enable mysql backend usage. |
| spring.cloud.vault.mysql.password-property | spring.datasource.password | Target property for the obtained username. |
| spring.cloud.vault.mysql.role |  | Role name for credentials. |
| spring.cloud.vault.mysql.username-property | spring.datasource.username | Target property for the obtained username. |
| spring.cloud.vault.port | 8200 | Vault server port. |
| spring.cloud.vault.postgresql.backend | postgresql | postgresql backend path. |
| spring.cloud.vault.postgresql.enabled | false | Enable postgresql backend usage. |
| spring.cloud.vault.postgresql.password-property | spring.datasource.password | Target property for the obtained username. |
| spring.cloud.vault.postgresql.role |  | Role name for credentials. |
| spring.cloud.vault.postgresql.username-property | spring.datasource.username | Target property for the obtained username. |
| spring.cloud.vault.rabbitmq.backend | rabbitmq | rabbitmq backend path. |
| spring.cloud.vault.rabbitmq.enabled | false | Enable rabbitmq backend usage. |
| spring.cloud.vault.rabbitmq.password-property | spring.rabbitmq.password | Target property for the obtained password. |
| spring.cloud.vault.rabbitmq.role |  | Role name for credentials. |
| spring.cloud.vault.rabbitmq.username-property | spring.rabbitmq.username | Target property for the obtained username. |
| spring.cloud.vault.read-timeout | 15000 | Read timeout; |
| spring.cloud.vault.scheme | https | Protocol scheme. Can be either "http" or "https". |
| spring.cloud.vault.ssl.cert-auth-path | cert | Mount path of the TLS cert authentication backend. |
| spring.cloud.vault.ssl.key-store |  | Trust store that holds certificates and private keys. |
| spring.cloud.vault.ssl.key-store-password |  | Password used to access the key store. |
| spring.cloud.vault.ssl.trust-store |  | Trust store that holds SSL certificates. |
| spring.cloud.vault.ssl.trust-store-password |  | Password used to access the trust store. |
| spring.cloud.vault.token |  | Static vault token. Required if {@link #authentication} is {@code TOKEN}. |
| spring.cloud.vault.uri |  | Vault URI. Can be set with scheme, host and port. |
| spring.cloud.zookeeper.base-sleep-time-ms | 50 | Initial amount of time to wait between retries |
| spring.cloud.zookeeper.block-until-connected-unit |  | The unit of time related to blocking on connection to Zookeeper |
| spring.cloud.zookeeper.block-until-connected-wait | 10 | Wait time to block on connection to Zookeeper |
| spring.cloud.zookeeper.connect-string | localhost:2181 | Connection string to the Zookeeper cluster |
| spring.cloud.zookeeper.default-health-endpoint |  | Default health endpoint that will be checked to verify that a dependency is alive |
| spring.cloud.zookeeper.dependencies |  | Mapping of alias to ZookeeperDependency. From Ribbon perspective the alias is actually serviceID since Ribbon can’t accept nested structures in serviceID |
| spring.cloud.zookeeper.dependency-configurations |  |  |
| spring.cloud.zookeeper.dependency-names |  |  |
| spring.cloud.zookeeper.discovery.enabled | true |  |
| spring.cloud.zookeeper.discovery.initial-status |  | The initial status of this instance (defaults to {@link StatusConstants#STATUS\_UP}). |
| spring.cloud.zookeeper.discovery.instance-host |  | Predefined host with which a service can register itself in Zookeeper. Corresponds to the {code address} from the URI spec. |
| spring.cloud.zookeeper.discovery.instance-id |  | Id used to register with zookeeper. Defaults to a random UUID. |
| spring.cloud.zookeeper.discovery.instance-port |  | Port to register the service under (defaults to listening port) |
| spring.cloud.zookeeper.discovery.instance-ssl-port |  | Ssl port of the registered service. |
| spring.cloud.zookeeper.discovery.metadata |  | Gets the metadata name/value pairs associated with this instance. This information is sent to zookeeper and can be used by other instances. |
| spring.cloud.zookeeper.discovery.register | true | Register as a service in zookeeper. |
| spring.cloud.zookeeper.discovery.root | /services | Root Zookeeper folder in which all instances are registered |
| spring.cloud.zookeeper.discovery.uri-spec | {scheme}://{address}:{port} | The URI specification to resolve during service registration in Zookeeper |
| spring.cloud.zookeeper.enabled | true | Is Zookeeper enabled |
| spring.cloud.zookeeper.max-retries | 10 | Max number of times to retry |
| spring.cloud.zookeeper.max-sleep-ms | 500 | Max time in ms to sleep on each retry |
| spring.cloud.zookeeper.prefix |  | Common prefix that will be applied to all Zookeeper dependencies' paths |
| spring.integration.poller.fixed-delay | 1000 | Fixed delay for default poller. |
| spring.integration.poller.max-messages-per-poll | 1 | Maximum messages per poll for the default poller. |
| spring.sleuth.annotation.enabled | true |  |
| spring.sleuth.async.configurer.enabled | true | Enable default AsyncConfigurer. |
| spring.sleuth.async.enabled | true | Enable instrumenting async related components so that the tracing information is passed between threads. |
| spring.sleuth.baggage-keys |  | List of baggage key names that should be propagated out of process. These keys will be prefixed with baggage before the actual key. This property is set in order to be backward compatible with previous Sleuth versions. @see brave.propagation.ExtraFieldPropagation.FactoryBuilder#addPrefixedFields(String, java.util.Collection) |
| spring.sleuth.enabled | true |  |
| spring.sleuth.feign.enabled | true | Enable span information propagation when using Feign. |
| spring.sleuth.feign.processor.enabled | true | Enable post processor that wraps Feign Context in its tracing representations. |
| spring.sleuth.http.enabled | true |  |
| spring.sleuth.http.legacy.enabled | false |  |
| spring.sleuth.hystrix.strategy.enabled | true | Enable custom HystrixConcurrencyStrategy that wraps all Callable instances into their Sleuth representative - the TraceCallable. |
| spring.sleuth.integration.enabled | true | Enable Spring Integration sleuth instrumentation. |
| spring.sleuth.integration.patterns | [!hystrixStreamOutput\*, \*] | An array of patterns against which channel names will be matched. @see org.springframework.integration.config.GlobalChannelInterceptor#patterns(). Defaults to any channel name not matching the Hystrix Stream channel name. |
| spring.sleuth.integration.websockets.enabled | true | Enable tracing for WebSockets. |
| spring.sleuth.keys.http.headers |  | Additional headers that should be added as tags if they exist. If the header value is multi-valued, the tag value will be a comma-separated, single-quoted list. |
| spring.sleuth.keys.http.prefix | http. | Prefix for header names if they are added as tags. |
| spring.sleuth.log.slf4j.enabled | true | Enable a {@link Slf4jCurrentTraceContext} that prints tracing information in the logs. |
| spring.sleuth.messaging.enabled | false |  |
| spring.sleuth.messaging.kafka.enabled | false |  |
| spring.sleuth.messaging.kafka.remote-service-name | kafka |  |
| spring.sleuth.messaging.rabbit.enabled | false |  |
| spring.sleuth.messaging.rabbit.remote-service-name | rabbitmq |  |
| spring.sleuth.opentracing.enabled | true |  |
| spring.sleuth.propagation-keys |  | List of fields that are referenced the same in-process as it is on the wire. For example, the name "x-vcap-request-id" would be set as-is including the prefix. <p>Note: {@code fieldName} will be implicitly lower-cased. @see brave.propagation.ExtraFieldPropagation.FactoryBuilder#addField(String) |
| spring.sleuth.rxjava.schedulers.hook.enabled | true | Enable support for RxJava via RxJavaSchedulersHook. |
| spring.sleuth.rxjava.schedulers.ignoredthreads | [HystrixMetricPoller, ^RxComputation.\*$] | Thread names for which spans will not be sampled. |
| spring.sleuth.sampler.probability | 0.1 | Probability of requests that should be sampled. E.g. 1.0 - 100% requests should be sampled. The precision is whole-numbers only (i.e. there’s no support for 0.1% of the traces). |
| spring.sleuth.scheduled.enabled | true | Enable tracing for {@link org.springframework.scheduling.annotation.Scheduled}. |
| spring.sleuth.scheduled.skip-pattern | org.springframework.cloud.netflix.hystrix.stream.HystrixStreamTask | Pattern for the fully qualified name of a class that should be skipped. |
| spring.sleuth.supports-join | true | True means the tracing system supports sharing a span ID between a client and server. |
| spring.sleuth.trace-id128 | false | When true, generate 128-bit trace IDs instead of 64-bit ones. |
| spring.sleuth.web.additional-skip-pattern |  | Additional pattern for URLs that should be skipped in tracing. This will be appended to the {@link SleuthWebProperties#skipPattern} |
| spring.sleuth.web.client.enabled | true | Enable interceptor injecting into {@link org.springframework.web.client.RestTemplate} |
| spring.sleuth.web.enabled | true | When true enables instrumentation for web applications |
| spring.sleuth.web.skip-pattern | /api-docs.\* | /autoconfig |
| /configprops | /dump | /health |
| /info | /metrics.\* | /mappings |
| /trace | /swagger.\* | .\*\.png |
| .\*\.css | .\*\.js | .\*\.html |
| /favicon.ico | /hystrix.stream | /application/.\* |
| /actuator.\* | /cloudfoundryapplication | Pattern for URLs that should be skipped in tracing |
| spring.sleuth.zuul.enabled | true | Enable span information propagation when using Zuul. |
| stubrunner.amqp.enabled | false | Whether to enable support for Stub Runner and AMQP. |
| stubrunner.amqp.mockCOnnection | true | Whether to enable support for Stub Runner and AMQP mocked connection factory. |
| stubrunner.classifier | stubs | The classifier to use by default in ivy co-ordinates for a stub. |
| stubrunner.cloud.consul.enabled | true | Whether to enable stubs registration in Consul. |
| stubrunner.cloud.delegate.enabled | true | Whether to enable DiscoveryClient’s Stub Runner implementation. |
| stubrunner.cloud.enabled | true | Whether to enable Spring Cloud support for Stub Runner. |
| stubrunner.cloud.eureka.enabled | true | Whether to enable stubs registration in Eureka. |
| stubrunner.cloud.ribbon.enabled | true | Whether to enable Stub Runner’s Ribbon integration. |
| stubrunner.cloud.stubbed.discovery.enabled | true | Whether Service Discovery should be stubbed for Stub Runner. If set to false, stubs will get registered in real service discovery. |
| stubrunner.cloud.zookeeper.enabled | true | Whether to enable stubs registration in Zookeeper. |
| stubrunner.consumer-name |  | You can override the default {@code spring.application.name} of this field by setting a value to this parameter. |
| stubrunner.delete-stubs-after-test | true | If set to {@code false} will NOT delete stubs from a temporary folder after running tests |
| stubrunner.ids | [] | The ids of the stubs to run in "ivy" notation ([groupId]:artifactId:[version]:[classifier][:port]). {@code groupId}, {@code classifier}, {@code version} and {@code port} can be optional. |
| stubrunner.ids-to-service-ids |  | Mapping of Ivy notation based ids to serviceIds inside your application Example "a:b" → "myService" "artifactId" → "myOtherService" |
| stubrunner.integration.enabled | true | Whether to enable Stub Runner integration with Spring Integration. |
| stubrunner.mappings-output-folder |  | Dumps the mappings of each HTTP server to the selected folder |
| stubrunner.max-port | 15000 | Max value of a port for the automatically started WireMock server |
| stubrunner.min-port | 10000 | Min value of a port for the automatically started WireMock server |
| stubrunner.password |  | Repository password |
| stubrunner.properties |  | Map of properties that can be passed to custom {@link org.springframework.cloud.contract.stubrunner.StubDownloaderBuilder} |
| stubrunner.proxy-host |  | Repository proxy host |
| stubrunner.proxy-port |  | Repository proxy port |
| stubrunner.snapshot-check-skip | false | If set to {@code true} will not assert whether the downloaded stubs / contract JAR was downloaded from a remote location or a local one(only applicable to Maven repos, not Git or Pact) |
| stubrunner.stream.enabled | true | Whether to enable Stub Runner integration with Spring Cloud Stream. |
| stubrunner.stubs-mode |  | Pick where the stubs should come from |
| stubrunner.stubs-per-consumer | false | Should only stubs for this particular consumer get registered in HTTP server stub. |
| stubrunner.username |  | Repository username |