[Introduction](https://cloud.spring.io/spring-cloud-static/spring-cloud-function/2.1.1.RELEASE/spring-cloud-function.html" \l "_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](https://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.

[Building and Running a Function](https://cloud.spring.io/spring-cloud-static/spring-cloud-function/2.1.1.RELEASE/spring-cloud-function.html#_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.

Functions can also 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. There is also support for Message<Pojo> where the message headers are copied from the incoming event, depending on the adapter. The web adapter also supports conversion from form-encoded data to a Map, and if you are using the function with Spring Cloud Stream then all the conversion and coercion features for message payloads will be applicable as well.

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.

[Function Catalog and Flexible Function Signatures](https://cloud.spring.io/spring-cloud-static/spring-cloud-function/2.1.1.RELEASE/spring-cloud-function.html#_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, while providing a consistent execution model. That’s why all user defined functions are transformed into a canonical representation by FunctionCatalog, using primitives defined by the [Project Reactor](https://projectreactor.io/) (i.e., Flux<T> and Mono<T>). 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>>.

Using Reactor based primitives not only helps with the canonical representation of user defined functions, but it also facilitates a more robust and flexible(reactive) execution model.

While users don’t normally have to care about the FunctionCatalog at all, it is useful to know what kind of functions are supported in user code.

[Java 8 function support](https://cloud.spring.io/spring-cloud-static/spring-cloud-function/2.1.1.RELEASE/spring-cloud-function.html#_java_8_function_support)

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.

[Function Routing](https://cloud.spring.io/spring-cloud-static/spring-cloud-function/2.1.1.RELEASE/spring-cloud-function.html#_function_routing)

Since version 2.2 Spring Cloud Function provides routing feature allowing you to invoke a single function which acts as a router to an actual function you wish to invoke This feature is very useful in certain FAAS environments where maintaining configurations for several functions could be cumbersome or exposing more then one function is not possible.

You enable this feature via spring.cloud.function.routing.enabled property setting it to true (default is false). This enables RoutingFunction under the name router which is loaded in FunctionCatalog.

This function has the following signature:

public class RoutingFunction implements Function<Publisher<Message<?>>, Publisher<?>>, Consumer<Publisher<Message<?>>> {

. . .

}

This allows the above function to act as both Function and Consumer. As you can see it takes Message<?> as an input argument. This allows you to communicate the name of the actual function you want to invoke by providing function.name Message header.

In specific execution environments/models the adapters are responsible to translate and communicate function.name via Message header. For example, when using *spring-cloud-function-web* you can provide function.name as an HTTP header and the framework will propagate it as well as other HTTP headers as Message headers.

Using Message also allows us to benefit from `MessageConverter`s to convert incoming request to the actual input type of the target function

[Kotlin Lambda support](https://cloud.spring.io/spring-cloud-static/spring-cloud-function/2.1.1.RELEASE/spring-cloud-function.html#_kotlin_lambda_support)

We also provide support for Kotlin lambdas (since v2.0). Consider the following:

@Bean

open fun kotlinSupplier(): () -> String {

return { "Hello from Kotlin" }

}

@Bean

open fun kotlinFunction(): (String) -> String {

return { it.toUpperCase() }

}

@Bean

open fun kotlinConsumer(): (String) -> Unit {

return { println(it) }

}

The above represents Kotlin lambdas configured as Spring beans. The signature of each maps to a Java equivalent of Supplier, Function and Consumer, and thus supported/recognized signatures by the framework. While mechanics of Kotlin-to-Java mapping are outside of the scope of this documentation, it is important to understand that the same rules for signature transformation outlined in "Java 8 function support" section are applied here as well.

To enable Kotlin support all you need is to add spring-cloud-function-kotlin module to your classpath which contains the appropriate autoconfiguration and supporting classes.

[Standalone Web Applications](https://cloud.spring.io/spring-cloud-static/spring-cloud-function/2.1.1.RELEASE/spring-cloud-function.html" \l "_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 dependencies 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 a single function (consumer etc.) in the catalog, 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).

For cases where there is more then a single function in catalog and you want to map a specific function to the root path (e.g., "/"), or you want to compose several functions and then map to the root path you can do so by providing spring.cloud.function.definition property which essentially used by spring-=cloud-function-web module to provide default mapping for cases where there is some type of a conflict (e.g., more then one function available etc).

For example,

--spring.cloud.function.definition=foo|bar

The above property will compose 'foo' and 'bar' function and map the composed function to the "/" path.

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).

[Standalone Streaming Applications](https://cloud.spring.io/spring-cloud-static/spring-cloud-function/2.1.1.RELEASE/spring-cloud-function.html#_standalone_streaming_applications)

To send or receive messages from a broker (such as RabbitMQ or Kafka) you can leverage spring-cloud-stream project and it’s integration with Spring Cloud Function. Please refer to [Spring Cloud Function](https://docs.spring.io/spring-cloud-stream/docs/current/reference/htmlsingle/#_spring_cloud_function) section of the Spring Cloud Stream reference manual for more details and examples.

[Deploying a Packaged Function](https://cloud.spring.io/spring-cloud-static/spring-cloud-function/2.1.1.RELEASE/spring-cloud-function.html#_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).

[Functional Bean Definitions](https://cloud.spring.io/spring-cloud-static/spring-cloud-function/2.1.1.RELEASE/spring-cloud-function.html#_functional_bean_definitions)

Spring Cloud Function supports a "functional" style of bean declarations for small apps where you need fast startup. The functional style of bean declaration was a feature of Spring Framework 5.0 with significant enhancements in 5.1.

[Comparing Functional with Traditional Bean Definitions](https://cloud.spring.io/spring-cloud-static/spring-cloud-function/2.1.1.RELEASE/spring-cloud-function.html#_comparing_functional_with_traditional_bean_definitions)

Here’s a vanilla Spring Cloud Function application from with the familiar @Configuration and @Bean declaration style:

@SpringBootApplication

public class DemoApplication {

@Bean

public Function<String, String> uppercase() {

return value -> value.toUpperCase();

}

public static void main(String[] args) {

SpringApplication.run(DemoApplication.class, args);

}

}

You can run the above in a serverless platform, like AWS Lambda or Azure Functions, or you can run it in its own HTTP server just by including spring-cloud-function-starter-web on the classpath. Running the main method would expose an endpoint that you can use to ping that uppercase function:

$ curl localhost:8080 -d foo

FOO

The web adapter in spring-cloud-function-starter-web uses Spring MVC, so you needed a Servlet container. You can also use Webflux where the default server is netty (even though you can still use Servlet containers if you want to) - just include the spring-cloud-starter-function-webflux dependency instead. The functionality is the same, and the user application code can be used in both.

Now for the functional beans: the user application code can be recast into "functional" form, like this:

@SpringBootConfiguration

public class DemoApplication implements ApplicationContextInitializer<GenericApplicationContext> {

public static void main(String[] args) {

FunctionalSpringApplication.run(DemoApplication.class, args);

}

public Function<String, String> uppercase() {

return value -> value.toUpperCase();

}

@Override

public void initialize(GenericApplicationContext context) {

context.registerBean("demo", FunctionRegistration.class,

() -> new FunctionRegistration<>(uppercase())

.type(FunctionType.from(String.class).to(String.class)));

}

}

The main differences are:

* The main class is an ApplicationContextInitializer.
* The @Bean methods have been converted to calls to context.registerBean()
* The @SpringBootApplication has been replaced with @SpringBootConfiguration to signify that we are not enabling Spring Boot autoconfiguration, and yet still marking the class as an "entry point".
* The SpringApplication from Spring Boot has been replaced with a FunctionalSpringApplication from Spring Cloud Function (it’s a subclass).

The business logic beans that you register in a Spring Cloud Function app are of type FunctionRegistration. This is a wrapper that contains both the function and information about the input and output types. In the @Bean form of the application that information can be derived reflectively, but in a functional bean registration some of it is lost unless we use a FunctionRegistration.

An alternative to using an ApplicationContextInitializer and FunctionRegistration is to make the application itself implement Function (or Consumer or Supplier). Example (equivalent to the above):

@SpringBootConfiguration

public class DemoApplication implements Function<String, String> {

public static void main(String[] args) {

FunctionalSpringApplication.run(DemoApplication.class, args);

}

@Override

public String uppercase(String value) {

return value.toUpperCase();

}

}

It would also work if you add a separate, standalone class of type Function and register it with the SpringApplication using an alternative form of the run() method. The main thing is that the generic type information is available at runtime through the class declaration.

The app runs in its own HTTP server if you add spring-cloud-starter-function-webflux (it won’t work with the MVC starter at the moment because the functional form of the embedded Servlet container hasn’t been implemented). **The app also runs just fine in AWS Lambda or Azure Functions, and the improvements in startup time are dramatic.**

[Testing Functional Applications](https://cloud.spring.io/spring-cloud-static/spring-cloud-function/2.1.1.RELEASE/spring-cloud-function.html#_testing_functional_applications)

Spring Cloud Function also has some utilities for integration testing that will be very familiar to Spring Boot users. For example, here is an integration test for the HTTP server wrapping the app above:

@RunWith(SpringRunner.class)

@FunctionalSpringBootTest

@AutoConfigureWebTestClient

public class FunctionalTests {

@Autowired

private WebTestClient client;

@Test

public void words() throws Exception {

client.post().uri("/").body(Mono.just("foo"), String.class).exchange()

.expectStatus().isOk().expectBody(String.class).isEqualTo("FOO");

}

}

This test is almost identical to the one you would write for the @Bean version of the same app - the only difference is the @FunctionalSpringBootTest annotation, instead of the regular @SpringBootTest. All the other pieces, like the @Autowired WebTestClient, are standard Spring Boot features.

Or you could write a test for a non-HTTP app using just the FunctionCatalog. For example:

@RunWith(SpringRunner.class)

@FunctionalSpringBootTest

public class FunctionalTests {

@Autowired

private FunctionCatalog catalog;

@Test

public void words() throws Exception {

Function<Flux<String>, Flux<String>> function = catalog.lookup(Function.class,

"function");

assertThat(function.apply(Flux.just("foo")).blockFirst()).isEqualTo("FOO");

}

}

(The FunctionCatalog always returns functions from Flux to Flux, even if the user declares them with a simpler signature.)

[Limitations of Functional Bean Declaration](https://cloud.spring.io/spring-cloud-static/spring-cloud-function/2.1.1.RELEASE/spring-cloud-function.html#_limitations_of_functional_bean_declaration)

Most Spring Cloud Function apps have a relatively small scope compared to the whole of Spring Boot, so we are able to adapt it to these functional bean definitions easily. If you step outside that limited scope, you can extend your Spring Cloud Function app by switching back to @Bean style configuration, or by using a hybrid approach. If you want to take advantage of Spring Boot autoconfiguration for integrations with external datastores, for example, you will need to use @EnableAutoConfiguration. Your functions can still be defined using the functional declarations if you want (i.e. the "hybrid" style), but in that case you will need to explicitly switch off the "full functional mode" using spring.functional.enabled=false so that Spring Boot can take back control.

[Dynamic Compilation](https://cloud.spring.io/spring-cloud-static/spring-cloud-function/2.1.1.RELEASE/spring-cloud-function.html#_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

...

[Serverless Platform Adapters](https://cloud.spring.io/spring-cloud-static/spring-cloud-function/2.1.1.RELEASE/spring-cloud-function.html#_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.