



# API Toolbox with Spring

---

Spencer Gibb

@spencerbgibb 

[social.sdf.org/@spencergibb](https://social.sdf.org/@spencergibb) 

Tweet by James Smith from NounProject.com  
Mastodon by Icon 54 from iconscout.com

# Agenda

- HTTP
- GraphQL
- Messaging
- Testing
- Consuming APIs
- Q+A




Old wooden toolbox filled with work tools by Thomas Baker from Noun Project (CC BY-NC-ND 2.0)



**HTTP**

http by M. Oki Orlando from Noun Project (CC BY 3.0)

# Spring Web MVC

 <https://docs.spring.io/spring-framework/reference/web/webmvc.html>

 <https://docs.spring.io/spring-boot/docs/current/reference/html/web.html#web.servlet>

- Original Servlet-based Model-View-Controller framework
- Use **@Controller** or **@RestController** to create beans to handle incoming HTTP requests
- The **@RequestMapping** and friends (**@GetMapping**, etc.) are used to map HTTP Methods
- All Servlet interfaces can be registered as beans.
- Tomcat, Jetty, and Undertow
- Can use JAX-RS implementations as Spring MVC alternative.

# Spring Web MVC


```
@RestController
@RequestMapping("/users")
public class MyRestController {
    // fields and constructor omitted

    @GetMapping("/{userId}")
    public Mono<User> getUser(@PathVariable Long userId) {
        return userRepo.findById(userId);
    }

    @GetMapping("/{userId}/customers")
    public Flux<Customer> getUserCustomers(@PathVariable Long userId) {
        return userRepo.findById(userId).flatMapMany(customerRepo::findByUser);
    }

    @DeleteMapping("/{userId}")
    public Mono<Void> deleteUser(@PathVariable Long userId) {
        return userRepo.deleteById(userId);
    }
}
```

# Spring WebFlux

 <https://docs.spring.io/spring-framework/reference/web-reactive.html>

 <https://docs.spring.io/spring-boot/docs/current/reference/html/web.html#web.reactive>

- New asynchronous, non-blocking web framework based on the Project Reactor implementation of Reactive Streams
- Able to handle concurrency with a small number of threads and fewer hardware resources
- **Mono** is zero to one, and **Flux** is zero to many.
- Similar annotation style to Spring MVC
- Netty, Undertow, and the Tomcat and Jetty Servlet Containers

# Spring WebFlux

```
@RestController
@RequestMapping("/users")
public class MyRestController {
    // fields and constructor omitted

    @GetMapping("/{userId}")
    public Mono<User> getUser(@PathVariable Long userId) {
        return userRepo.findById(userId);
    }

    @GetMapping("/{userId}/customers")
    public Flux<Customer> getUserCustomers(@PathVariable Long userId) {
        return userRepo.findById(userId).flatMapMany(customerRepo::findByUser);
    }

    @DeleteMapping("/{userId}")
    public Mono<Void> deleteUser(@PathVariable Long userId) {
        return userRepo.deleteById(userId);
    }
}
```

# Spring WebMVC.fn

 <https://docs.spring.io/spring-framework/reference/web/webmvc-functional.html>

 <https://docs.spring.io/spring-boot/docs/current/reference/html/web.html#web.servlet>

- Lightweight functional programming model
- Functions are used to route and handle requests
- Alternative to annotation-based model of Spring MVC
- **RouterFunction** beans are created using the **RouterFunctions.route()** builder
- Same Servlet containers



# Spring WebMVC.fn

```
@Configuration(proxyBeanMethods = false)
public class MyRoutingConfiguration {

    private static final RequestPredicate ACCEPT_JSON =
        RequestPredicates.accept(MediaType.APPLICATION_JSON);

    @Bean
    public RouterFunction<ServerResponse> routerFunction(MyUserHandler users) {
        return RouterFunctions.route()
            .GET("/{user}", ACCEPT_JSON, users::getUser)
            .GET("/{user}/customers", ACCEPT_JSON, users::getUserCustomers)
            .DELETE("/{user}", ACCEPT_JSON, users::deleteUser)
            .build();
    }
}
```

# Spring WebFlux.fn

 <https://docs.spring.io/spring-framework/reference/web/webflux-functional.html>

 <https://docs.spring.io/spring-boot/docs/current/reference/html/web.html#web.reactive.webflux>


- Lightweight functional programming model
- Functions are used to route and handle requests
- Alternative to annotation-based model of Spring WebFlux
- **RouterFunction** beans are created using the **RouterFunctions.route()** builder
- Same reactive programming model

# Spring WebFlux.fn

```
@Configuration(proxyBeanMethods = false)
public class MyRoutingConfiguration {
    private static final RequestPredicate ACCEPT_JSON =
        RequestPredicates.accept(MediaType.APPLICATION_JSON);

    @Bean
    public RouterFunction<ServerResponse> monoRouterFunction(MyUserHandler users) {
        return RouterFunctions.route()
            .GET("/{user}", ACCEPT_JSON, users::getUser)
            .GET("/{user}/customers", ACCEPT_JSON, users::getUserCustomers)
            .DELETE("/{user}", ACCEPT_JSON, users::deleteUser)
            .build();
    }
}
```

# HTTP Content

 <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/MediaType.html>

 <https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-ann-async.html>

- JSON, XML, plain text, CBOR and many more defined in **MediaType**.
- Supports streaming with asynchronous requests using **DeferredResult** and **Callable**.
- Pluggable view technologies from Thymeleaf to PDFs.

# Spring Data Rest

 <https://spring.io/projects/spring-data>

 <https://docs.spring.io/spring-data/rest/reference/>

- Spring Data provides a consistent programming model for data access using a **Repository** abstraction.
- Relational support for JDBC & JPA
- Non-relational support for Redis, MongoDB, and Cassandra and community support for various IaaS databases
- Spring Data Rest exports Spring Data repositories as hypermedia-driven RESTful resources.
- Spring HATEOAS: Hypermedia as the engine of application state.

# Spring Data Rest

```
{ "_embedded": {  
  "employees": [  
    {  
      "id": 1,  
      "name": "Bilbo Baggins",  
      "role": "burglar",  
      "_links": {  
        "self": {  
          "href": "https://example.com:9001/employees/1"  
        },  
        "employees": {  
          "href": "https://example.com:9001/employees"  
        }  
      }  
    }  
  ]  
}
```

# Spring Cloud Gateway

 <https://docs.spring.io/spring-cloud-gateway/reference/index.html>

 <https://youtu.be/BEjfwYiu4RU> - Spring Cloud Gateway Recipes

 <https://youtu.be/UyxUkAagLFs> - Spring Cloud Gateway MVC

- API Gateway to aggregate microservices into one API
- Cross cutting concerns: security, resiliency, fault tolerance
- Compatible with Spring WebFlux and Spring MVC (new last year)
- Focus on developer experience
- Integration with other Spring Cloud projects:
  - Service Discovery, Load Balancer, Config, Circuit Breaker
- Commercial managed version on Cloud Foundry and Kubernetes



**GraphQL**



# GraphQL

**GraphQL** is a **query language** for your API and a server-side runtime for executing queries and mutations using a type system you **model** for your data. GraphQL isn't tied to any specific database or storage engine and is instead backed by your existing code and data.

# GraphQL Object Types

```
Object Type → type Product {  
  Field → id: ID! ← Non-nullable  
  Field → title: String ← Built-in scalar types  
  Field → desc: String  
}
```

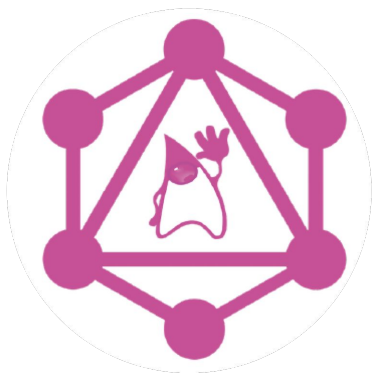
```
type Order {  
  id: ID!  
  product: Product ← Object Type  
  qty: Int  
  customer: Customer  
  orderedOn: Date ← Custom scalar type  
  status: OrderStatus ← Enum  
}
```

# GraphQL Object Types

```
Query → type Query {  
  allProducts: [Product] ← Return Type  
  getProduct(id: ID!): Product ← Argument  
  allOrders: [Order]  
  getOrder(id: ID!) : Order ← Required Argument  
  allCustomers: [Customer]  
  findCustomerByLastName(last: String!) : Customer  
}  
  
type Mutation {  
  createProduct(product: ProductInput) : Product  
}
```

The diagram illustrates the structure of GraphQL object types. It shows two types: `Query` and `Mutation`. The `Query` type contains several fields: `allProducts`, `getProduct`, `allOrders`, `getOrder`, `allCustomers`, and `findCustomerByLastName`. The `Mutation` type contains one field: `createProduct`. Annotations with arrows point to specific parts of the code: `Query` is labeled as the type name; `[Product]` in `allProducts` is labeled as the Return Type; `ID!` in `getProduct` is labeled as the Argument; `ID!` in `getOrder` is labeled as the Required Argument.

# GraphQL Libraries and Frameworks



**GraphQL Java**

<https://www.graphql-java.com>



**Spring for GraphQL**

<https://docs.spring.io/spring-graphql/reference/>



**DGS**

<https://netflix.github.io/dgs/>

# Spring for GraphQL

```
@Controller
public class BookController {
    @QueryMapping
    public Book bookById(@Argument Long id) {
        // ...
    }
    @MutationMapping
    public Book addBook(@Argument BookInput bookInput) {
        // ...
    }
    @SubscriptionMapping
    public Flux<Book> newPublications() {
        // ...
    }
}
```

**Dan Vega**

**Does Your API Need a REST?**

**Check Out GraphQL**

**Wed Feb 21 @ 10:00**

**ST-Laurent 1 (English)**



# Messaging

# Java Message Service (JMS)

 <https://docs.spring.io/spring-framework/reference/integration/jms.html>

 <https://docs.spring.io/spring-boot/docs/current/reference/html/messaging.html#messaging.jms>

- Provides a similar simplification for using JMS API, similar to Spring's JDBC integration
- Classic Spring Template type class: **JmsTemplate**.
- Spring Boot provides auto-configuration for ActiveMQ “Classic” and ActiveMQ Artemis



# Spring Cloud Function and Stream

 <https://docs.spring.io/spring-cloud-function/reference/index.html>

 <https://docs.spring.io/spring-cloud-stream/reference/index.html>

- Embraces core functional interfaces in Java defines as **@Bean**:
  - **Supplier<O>, Function<I, O>, Consumer<I>**
- Integration with IaaS serverless platforms and Spring Cloud Stream.
- Spring Cloud Stream is for building message-driven applications.
- Binders provide abstraction to popular middleware:
  - Kafka, RabbitMQ, Pulsar, AWS Kinesis, and others provided by the community

# WebSockets

 <https://docs.spring.io/spring-framework/reference/web/websocket.html>

 <https://docs.spring.io/spring-boot/docs/current/reference/html/messaging.html#messaging.websockets>

- RFC 6455, Provides a standardized way to establish a full-duplex, two-way communication channel between client and server over a single TCP connection
- Designed to work over HTTP on ports 80 and 443 for firewalls
- For Spring MVC, use **spring-boot-starter-websocket**
- It is included in **spring-boot-starter-webflux**

# RSocket

 <https://docs.spring.io/spring-framework/reference/rsocket.html>

 <https://docs.spring.io/spring-boot/docs/current/reference/html/messaging.html#messaging.rsocket>

- RSocket is an application protocol for multiplexed, duplex communication over TCP, WebSocket, & other byte stream transports.
- 4 interaction models: RR, RS, Channel, and F&F
- After initial connection, both sides can initiate interaction.
- Reactive Streams semantics across network boundary including back pressure.
- Much more, see <https://rsocket.io/about/protocol>



# Testing

# MockMVC

 <https://docs.spring.io/spring-framework/reference/testing/spring-mvc-test-framework.html>

 <https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.testing.spring-boot-applications.with-mock-environment>

- Testing support for Spring MVC applications
- Full Spring MVC request/response handling with mock implementations rather than a running server
- Integration with **WebTestClient**, which provides higher level abstractions instead of raw data as well as the option to switch to integration testing and running a server.

# MockMVC

```
@SpringBootTest
@AutoConfigureMockMvc
class MyMockMvcTests {
    @Test
    void testWithMockMvc(@Autowired MockMvc mvc) throws Exception {
        mvc.perform(get("/")).andExpect(status().isOk())
            .andExpect(content().string("Hello World"));
    }
    @Test
    void testWithWebTestClient(@Autowired WebTestClient webClient) {
        webClient.get().uri("/").exchange().expectStatus().isOk()
            .expectBody(String.class).isEqualTo("Hello World");
    }
}
```

# Wiremock

 <https://wiremock.org/>

 <https://docs.spring.io/spring-cloud-contract/docs/current/reference/html/project-features.html#features-wiremock>

- Wiremock allows mocking external APIs.
- Java or JSON configuration.
- Flexible request matching.
- Record/playback
- HTTP, gRPC, and GraphQL
- Add **spring-cloud-starter-contract-stub-runner** and **@AutoConfigureWireMock**.

# Wiremock

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@AutoConfigureWireMock(port = 0)
public class WiremockForDocsTests {
    @Autowired private Service service; // A service that calls out over HTTP
    @BeforeEach
    public void setup() {
        this.service.setBase("http://localhost:"
            + this.environment.getProperty("wiremock.server.port"));
    }
    @Test
    public void contextLoads() throws Exception {
        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!");
    }
}
```



# Spring Cloud Contract

 <https://docs.spring.io/spring-cloud-contract/docs/current/reference/html/getting-started.html>

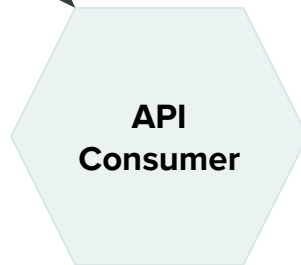
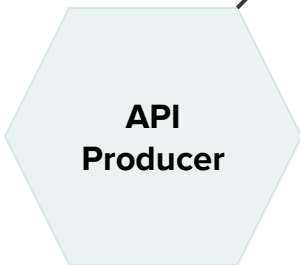
- Moves TDD to the level of Software Architecture.
- Provides consumer-driven and producer-driven contract testing.
- Various overloaded methods for combinations of HTTP method, return type, and parameters
- Ensures that HTTP and messaging stubs (used when developing the client) do exactly what the actual server-side implementation does.
- Provides the ability to publish contract changes for two way visibility (producer and consumer)

# Spring Cloud Contract


During testing, Producer creates stubs and publishes them to Artifact Repository



During testing, Consumer pulls stubs from Artifact Repository and verifies no breaking changes have occurred



# Testcontainers

 <https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.testing.testcontainers>


 <https://testcontainers.com/>

- The Testcontainers library provides a way to manage services running inside Docker containers during testing.
- Supports native **@Testcontainers** & **@Container** annotations.
- The Spring Boot **@ServiceConnection** annotation bridges metadata (host, ports, etc.) from Testcontainers to Spring Boot configuration.
- Support for many technologies by default, including: databases, message brokers, cloud services, WireMock, and more.



# Consuming APIs

# RestTemplate

 <https://docs.spring.io/spring-framework/reference/integration/rest-clients.html#rest-resttemplate>

 <https://docs.spring.io/spring-boot/docs/current/reference/html/io.html#io.rest-client.resttemplate>

- The classic HTTP Client offered by Spring Framework.
- Classic Spring Template type class.
- Various overloaded methods for combinations of HTTP method, return type, and parameters

# RestTemplate

```
@Service
public class MyService {
    private final RestTemplate restTemplate;

    public MyService(RestTemplateBuilder restTemplateBuilder) {
        restTemplate = restTemplateBuilder.build();
    }

    public Details someRestCall(String name) {
        return restTemplate.getForObject("/{name}/details",
            Details.class, name);
    }
}
```

# WebClient

 <https://docs.spring.io/spring-framework/reference/integration/rest-clients.html#rest-webclient>

 <https://docs.spring.io/spring-boot/docs/current/reference/html/io.html#io.rest-client.webclient>

- Non-blocking, reactive HTTP Client introduced in Framework 5.0.
- Reactive Streams back pressure.
- Functional-style fluent API.
- Requires WebFlux (even in a Spring MVC app).

# WebClient

```
@Service
public class MyService {
    private final WebClient webClient;

    public MyService(WebClient.Builder webClientBuilder) {
        this.webClient = webClientBuilder.baseUrl("https://example.org").build();
    }

    public Mono<Details> someRestCall(String name) {
        return this.webClient.get().uri("/{name}/details", name)
            .retrieve().bodyToMono(Details.class);
    }
}
```



# RestClient

 <https://docs.spring.io/spring-framework/reference/integration/rest-clients.html#rest-restclient>

 <https://docs.spring.io/spring-boot/docs/current/reference/html/io.html#io.rest-client.restclient>

- Synchronous HTTP Client introduced in Framework 6.1.
- Functional-style fluent API (similar to WebClient)
- WebFlux not required
- Same underlying configuration as RestTemplate


# RestClient

```
@Service
public class MyService {
    private final RestClient restClient;

    public MyService(RestClient.Builder restClientBuilder) {
        this.restClient = restClientBuilder.baseUrl("https://example.org").build();
    }

    public Details someRestCall(String name) {
        return this.restClient.get().uri("/{name}/details", name)
            .retrieve().body(Details.class);
    }
}
```

# Interface Clients

 <https://docs.spring.io/spring-framework/reference/integration/rest-clients.html#rest-http-interface>

 <https://github.com/spring-projects/spring-boot/issues/31337>

- Declarative web client
- Annotate a Java interface and the library provides the implementation
- Originally supported OpenFeign in Spring Cloud
- Added to Spring Framework in 6.0
- Auto-configuration still to come in Spring Boot
- **@HttpExchange** and **@RSocketExchange**.
- **@HttpExchange** can use **WebClient**, **RestClient**, or **RestTemplate** .

# Interface Clients: @HttpExchange

```
public interface VerificationService {
    @PostExchange(url = "/verify") // meta-annotated @HttpExchange(method = "POST")
    VerifyResult verify(@RequestBody CustomerApplication customerApplication);
}

@Bean
public VerificationService vs(RestClient.Builder builder, ConversionService cs) {
    RestClient rc = builder.baseUrl(this.verificationServiceUrl).build();
    // in the future, this will be auto-configured by Spring Boot
    HttpServiceProxyFactory hspf = HttpServiceProxyFactory.builderFor(
        RestClientAdapter.create(rc)).conversionService(cs).build();
    return hspf.createClient(VerificationService.class);
}

// somewhere else in the application
VerifyResult result = verificationService.verify(customerApplication);
```

# Interface Clients: @RSocketExchange

```
public interface RadarsService {  
    @RSocketExchange("locate.radars.within")  
    Flux<AirportLocation> radars(MapRequest request);  
}
```

@Controller

```
public class RadarsController implements RadarsService {  
    public Flux<AirportLocation> radars(MapRequest request) {  
        // ...  
    }  
}
```



# Documenting APIs

# Spring REST Docs

 <https://docs.spring.io/spring-restdocs/docs/current/reference/htmlsingle/>

 <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#features.testing.spring-boot-applications.autoconfigured-spring-restdocs>

- Uses AsciiDoctor by default
- Produces snippets by testing with MockMvc, WebTestClient, or Rest Assured
- The snippets are accurate if the tests pass.
- Spring Cloud Contract integration

```
[[resources_index_access]]  
=== Accessing the index
```

A `GET` request is used to access the index

```
operation::index-example[snippets='response-fields,http-response,links']
```

# Questions



# Thank you

<https://gibb.tech/preso/confoo-2024-api-toolbox-with-spring/>

@spencerbgibb 

social.sdf.org/@spencergibb 

Question by Mani Cheng from NounProject.com  
Tweet by James Smith from NounProject.com  
Mastodon by Icon 54 from iconscout.com

