Spring Recipes: A Collection of Common-Sense Solutions

Nathaniel Schutta (@ntschutta)
ntschutta.io
Dan Vega (@therealdanvega)
danvega.dev

https://github.com/spring-recipes/recipes

spring ACADEMY

Our Vision

Empower Spring developers to continuously learn, innovate, and solve complex problems.

Core Features

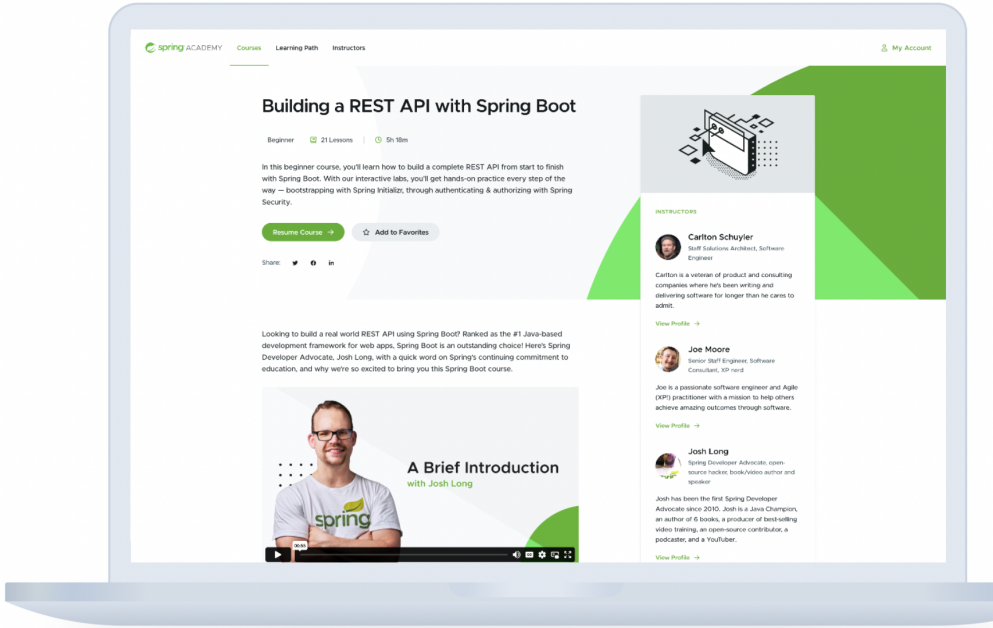
- Project-based courses with hands-on labs
- In-browser code editing and debugging
- Certification prep courses
- Spring Certified Professional Certification

Sign Up Today

Get started for free at https://spring.academy

For Conference Attendees:

Use **DEVNEXUS** at checkout by April 10th for a **20%** discount on a Spring Academy Pro subscription!



Who are these guys?

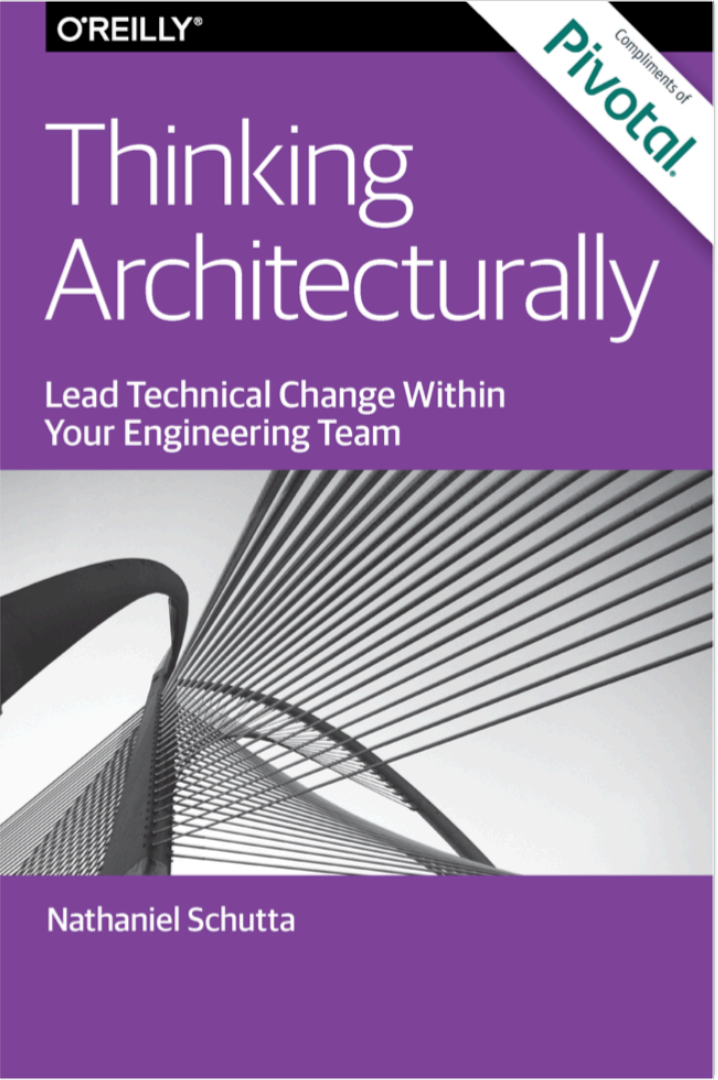
ABOUT ME

- Husband & Father
- Twin Cites, MN
- Software Development 20+ Years
- Architect as a Service
- Developer Advocate
- Author
- Adjunct Professor
- Blogger, Twitch, YouTube
- Golfer, cyclist
- @ntschutta
- http://ntschutta.io

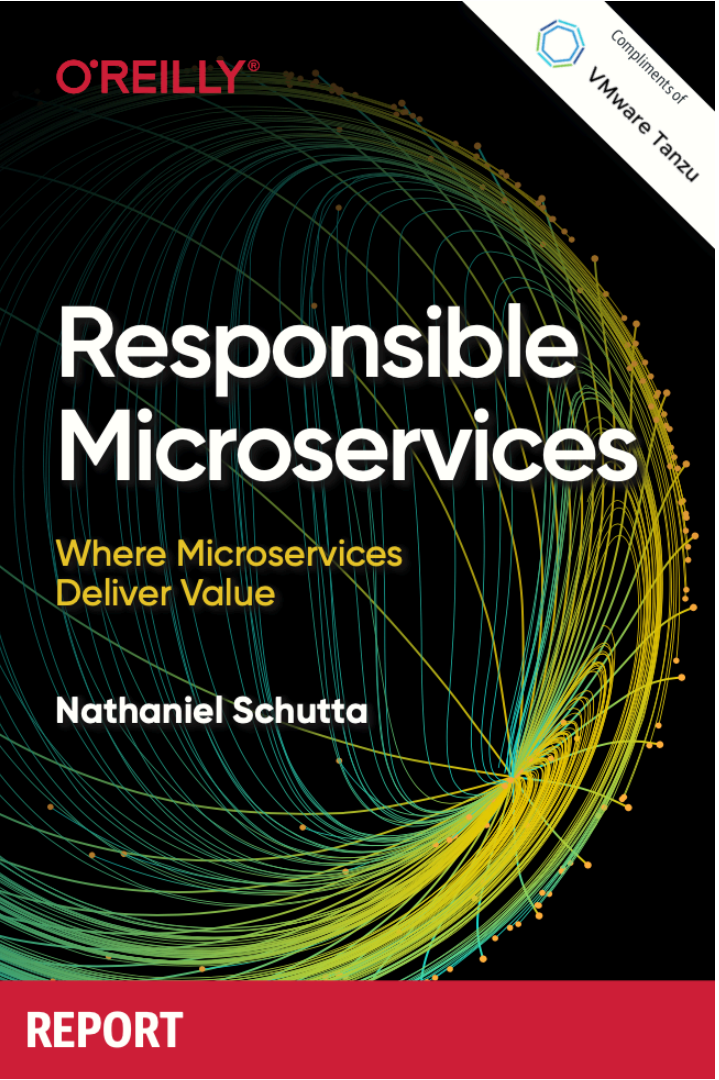
Between Chair and Keyboard

Most Mondays,
around noon Central
https://www.twitch.tv/vmwaretanzu

Nate Schutta
Software Architect
VMware
@ntschutta



https://tanzu.vmware.com/content/ebooks/thinking-architecturally



https://tanzu.vmware.com/content/ebooks/responsible-microservices-ebook

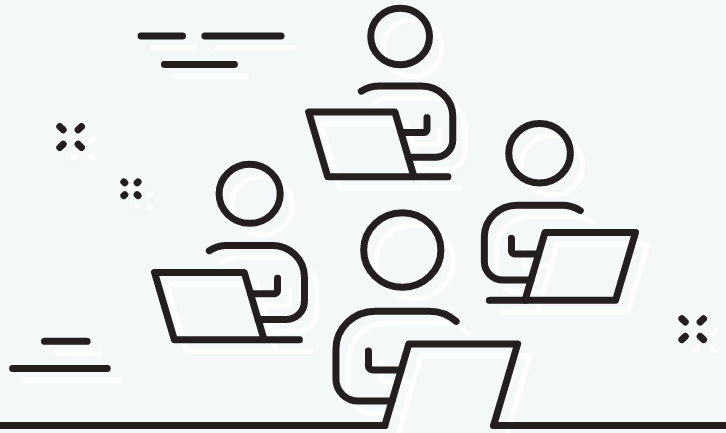
ABOUT ME

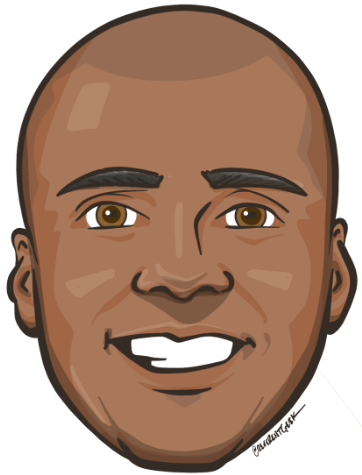
- Husband & Father
- Cleveland, OH
- Software Development 20+ Years
- Spring Developer Advocate
- Content Creator (www.danvega.dev)
  - Blogger
  - YouTuber
  - Course Creator
- @therealdanvega

spring

OFFICE HOURS

https://bit.ly/spring-office-hours





https://tanzu.vmware.com/developer/tv/spring-office-hours/


spring®
RECIPES


Java™


spring®

Spring Projects


Spring Boot


Spring Cloud


Spring Framework

Spring Cloud Data Flow

Spring Tool Suite


Spring LDAP


Spring Cloud Gateway


Spring Security


Spring Data


Spring Batch


Spring Integration


Project Reactor


Spring Kafka


Spring for GraphQL


Spring Web Services


Spring Web Flow


Spring Hateoas


Spring AMQP

Agenda

- Getting Started with Spring Boot
- Building Web Applications
- Working with Databases
- Spring Cloud
- Spring Security
- Testing
- Production
- Q+A

Getting Started with Spring Boot

Problem

You want to upgrade your application to Spring 6/Boot 3 but your team/architect/manager is skeptical.

Solution

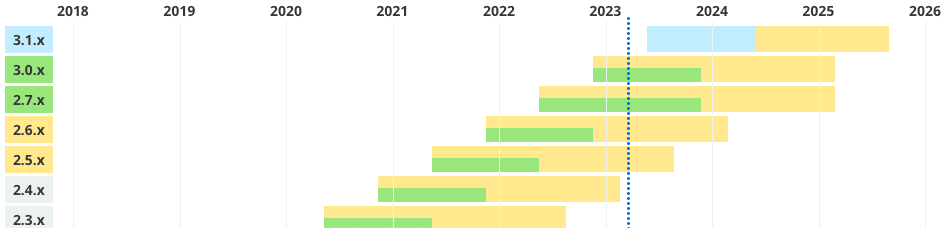
Features and functionality! Baseline support for Java 17 and related language features! Native executables! Spring Observability! Support! Without a commercial support license, Spring Boot 2.7.x support ends in November of 2023 while Spring 5.3.x support will end in December of 2024.

- Spring Boot
- Spring Framework
- Spring Data
- Spring Cloud
- Spring Cloud Data Flow
- Spring Security
- Spring Authorization Server
- Spring for GraphQL
- Spring Session
- Spring Integration
- Spring HATEOAS
- Spring REST Docs
- Spring Batch
- Spring AMQP
- Spring CredHub
- Spring Flo
- Spring for Apache Kafka
- Spring LDAP
- Spring Shell
- Spring Statemachine
- Spring Vault
- Spring Web Flow
- Spring Web Services

Spring Boot 3.0.5

OVERVIEW | LEARN | SUPPORT | SAMPLES

Branch	Initial Release	End of Support	End Commercial Support *
3.1.x	2023-05-18	2024-05-18	2025-08-18
3.0.x	2022-11-24	2023-11-24	2025-02-24
2.7.x	2022-05-19	2023-11-18	2025-02-18
2.6.x	2021-11-17	2022-11-24	2024-02-24
2.5.x	2021-05-20	2022-05-19	2023-08-24
2.4.x	2020-11-12	2021-11-18	2023-02-23
2.3.x	2020-05-15	2021-05-20	2022-08-20
2.2.x	2019-10-16	2020-10-16	2022-01-16
2.1.x	2018-10-30	2019-10-30	2021-01-30
2.0.x	2018-03-01	2019-03-01	2020-06-01
1.5.x	2017-01-30	2019-08-06	2020-11-06

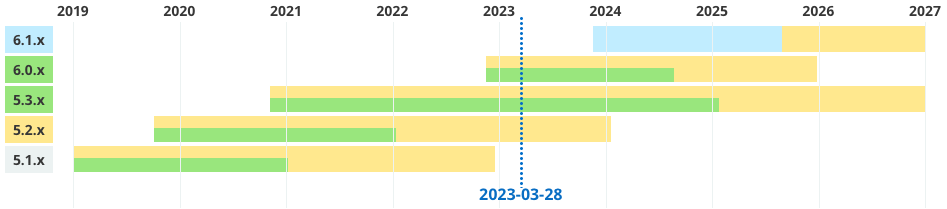


- Spring Boot
- Spring Framework
- Spring Data
- Spring Cloud
- Spring Cloud Data Flow
- Spring Security
- Spring Authorization Server
- Spring for GraphQL
- Spring Session
- Spring Integration
- Spring HATEOAS
- Spring REST Docs
- Spring Batch
- Spring AMQP
- Spring CredHub
- Spring Flo
- Spring for Apache Kafka
- Spring LDAP
- Spring Shell
- Spring Statemachine
- Spring Vault
- Spring Web Flow
- Spring Web Services

Spring Framework 6.0.7

OVERVIEW LEARN SUPPORT

Branch	Initial Release	End of Support	End Commercial Support *
6.1.x	2023-11-15	2025-08-31	2026-12-31
6.0.x	2022-11-16	2024-08-31	2025-12-31
5.3.x	2020-10-27	2024-12-31	2026-12-31
5.2.x	2019-09-30	2021-12-31	2023-12-31
5.1.x	2018-09-21	2020-12-31	2022-12-31



OSS support

Free security updates and bugfixes with support from the Spring community. See VMware Tanzu OSS support policy.

Commercial support

Business support from Spring experts during the OSS timeline, plus extended support after OSS End-Of-Life.

Publicly available releases for critical bugfixes and security issues when requested by customers.

Future release

Generation not yet released, timeline is subject to changes.

Work in small steps.

Avoid big bang changes.

Problem

You want to upgrade your application to Spring 6/Boot 3 but you aren't sure how to go about the process.

Solution

Upgrade to Java 17. Upgrade to the latests 2.7.x release. Check your project dependencies. Some configuration properties have changed, leverage `spring-boot-properties-migrator`. Some `import` statements will need to change to comply with Jakarta EE 10.

Spring Boot 3.0 Migration Guide

Andy Wilkinson edited this page 2 weeks ago · 25 revisions

This document is meant to help you migrate your application to Spring Boot 3.0.

Before You Start

Upgrade to the Latest `2.7.x` Version

Before you start the upgrade, make sure to upgrade to the latest available `2.7.x` version. This will make sure that you are building against the most recent dependencies of that line.

Review Dependencies

The move to Spring Boot 3 will upgrade a number of dependencies and might require work on your end. You can review dependency management for `2.7.x` with dependency management for `3.0.x` to asses how your project is affected.

You may also use dependencies that are not managed by Spring Boot (e.g. Spring Cloud). As your project defines an explicit version for those, you need first to identify the compatible version before upgrading.

Spring Security

Spring Boot 3.0 uses Spring Security 6.0. The Spring Security team have released Spring Security 5.8 to simplify upgrading to Spring Security 6.0. Before upgrading to Spring Boot 3.0, consider upgrading your Spring Boot 2.7 application to Spring Security 5.8. The Spring Security team have produced a migration guide that will help you to do so. From there, you can follow the 5.8 to 6.0 migration guide when upgrading to Spring Boot 3.0.

Dispatch types

In Servlet applications, Spring Security 6.0 applies authorization to every dispatch type. To align with this Spring Boot now configures Spring Security's filter to be called for every dispatch type. The types can be configured using the `spring.security.filter.dispatcher-types` property.

Review System Requirements

Upgrading to Spring Framework 6.x

Rossen Stoyanchev edited this page 3 weeks ago · 35 revisions

This page provides guidance on upgrading to Spring Framework 6.0.

Upgrading to Version 6.0

Core Container

The JSR-330 based `@Inject` annotation is to be found in `jakarta.inject` now. The corresponding JSR-250 based annotations `@PostConstruct` and `@PreDestroy` are to be found in `jakarta.annotation`. For the time being, Spring keeps detecting their `javax` equivalents as well, covering common use in pre-compiled binaries.

The core container performs basic bean property determination without `java.beans.Introspector` by default. For full backwards compatibility with 5.3.x in case of sophisticated JavaBeans usage, specify the following content in a `META-INF/spring.factories` file which enables 5.3-style full `java.beans.Introspector` usage:
`org.springframework.beans.BeanInfoFactory=org.springframework.beans.ExtendedBeanInfoFactory`

When staying on 5.3.x for the time being, you may enforce forward compatibility with 6.0-style property determination (and better introspection performance!) through a custom `META-INF/spring.factories` file:
`org.springframework.beans.BeanInfoFactory=org.springframework.beans.SimpleBeanInfoFactory`

`LocalVariableTableParameterNameDiscoverer` is deprecated now and logs a warning for each successful resolution attempt (it only kicks in when `StandardReflectionParameterNameDiscoverer` has not found names). Compile your Java sources with the common Java 8+ `-parameters` flag for parameter name retention (instead of relying on the `-debug` compiler flag) in order to avoid that warning, or report it to the maintainers of the affected code. With the Kotlin compiler, we recommend the `-java-parameters` flag for completeness.

`LocalValidatorFactoryBean` relies on standard parameter name resolution in Bean Validation 3.0 now, just configuring additional Kotlin reflection if Kotlin is present. If you refer to parameter names in your Bean Validation setup, make sure to compile your Java sources with the Java 8+ `-parameters` flag.

`ListenableFuture` has been deprecated in favor of `CompletableFuture`. See 27780.

Methods annotated with `@Async` must return either `Future` or `void`. This has long been documented, but is now also

Migrate to Jakarta EE 9

org.openrewrite.java.migrate.jakarta.JavaxMigrationToJakarta *Jakarta EE 9 is the first version of Jakarta EE that uses the new `jakarta` namespace.*

Tags

- jaxb
- jaxws
- jakarta

Source

Github, Issue Tracker, Maven Central

- groupId: org.openrewrite.recipe
- artifactId: rewrite-migrate-java
- version: 1.18.0

Usage

This recipe has no required configuration options. It can be activated by adding a dependency on `org.openrewrite.recipe:rewrite-migrate-java:1.18.0` in your build file or by running a shell command (in which case no build changes are needed):

Gradle | Maven POM | Maven Command Line

build.gradle

```
plugins {
    id("org.openrewrite.rewrite") version("5.38.0")
}
```

Early Access Program IntelliJ IDEA

IntelliJ IDEA EAP 6: Automatic Migration From Java EE to Jakarta EE, Shared Indexes for Spring Boot Projects, and More

Irina Maryasova

June 29, 2021

IntelliJ IDEA 2021.2 EAP 6 is now available! You can get the new build from our website, through the free Toolbox App, or as a snap (for Ubuntu).

In this final EAP before Beta, we have focused on some framework-specific updates. These include a new refactoring to simplify and speed up the migration of Java EE usages to Jakarta EE, automatic download of shared indexes for new Spring Boot projects, microservices diagrams, and more. This build also conveniently groups on-save actions in one place and offers a reworked structure view for diagrams.

Let's learn more about the new features.

Frameworks and technologies

Work in small steps.

Avoid big bang changes.

Building Web Applications

Problem

I need to validate the data that a client is sending to our REST APIs.

Solution

The Spring Framework provides support for the Java Bean Validation API.

```java
@PostMapping
public Coffee create(@RequestBody Coffee coffee) {
    return coffeeService.create(coffee);
}
```

```
POST http://localhost:8080/api/coffee
Content-Type: application/json

{
  "id": null,
  "name": "Cafè Latte",
  "size": "SHORT",
  "price": 3.99,
  "cost": 1.99
}
```

```java
@PostMapping
public Coffee create(@RequestBody Coffee coffee) {
    return coffeeService.create(coffee);
}
```

```
POST http://localhost:8080/api/coffee
Content-Type: application/json

{
  "id": null,
  "name": "",
  "size": "SHORT",
  "price": 199.99,
  "cost": 1.99
}
```

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

```java
import jakarta.validation.Valid;



@PostMapping
public Coffee create(@Valid @RequestBody Coffee coffee) {
    return coffeeService.create(coffee);
}
```

```java
package dev.danvega.javabucksrest.model;

import jakarta.validation.constraints.Max;
import jakarta.validation.constraints.NotEmpty;
import java.math.BigDecimal;

public record Coffee(
        Integer id,
        @NotEmpty
        String name,
        Size size,
        @Max(10)
        BigDecimal price,
        BigDecimal cost) {
}
```

```
{
  "timestamp": "2022-08-03T14:00:30.111+00:00",
  "status": 400,
  "error": "Bad Request",
  "message": "Validation failed for object='coffee'. Error count: 2",
  "path": "/api/coffee"
}
```

```
server.error.include-message=always
server.error.include-binding-errors=always
```

```
{
  "timestamp": "2022-08-03T13:59:36.327+00:00",
  "status": 400,
  "error": "Bad Request",
  "message": "Validation failed for object='coffee'. Error count: 2",
  "errors": [
    {
      "codes": [
        "Max.coffee.price",
        "Max.price",
        "Max.java.math.BigDecimal",
        "Max"
      ],
      "arguments": [
        {
          "codes": [
            "coffee.price",
            "price"
          ],
          "arguments": null,
          "defaultMessage": "price",
          "code": "price"
        },
        10
      ],
      "defaultMessage": "must be less than or equal to 10",
      "objectName": "coffee",
      "field": "price",
      "rejectedValue": 11.99,
      "bindingFailure": false,
      "code": "Max"
```

Problem

I’m interested in what problems GraphQL solves but I’m not sure how to build GraphQL APIs using the tools I already have.

Solution

Spring for GraphQL provides support for Spring applications built on GraphQL Java.

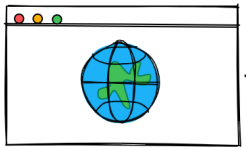
- Java (server) implementation for GraphQL
- Started in 2015 by Andreas Marek
- Pure Execution Engine: No HTTP or IO

GraphQL Java

- Joint collaboration by both teams
- Supports GraphQL requests over
    - HTTP
    - WebSocket
    - RSocket
- Make complex problems easy

Spring
for GraphQL

Client & Server 🤝



Client

```
query {
  findCustomerByLastName(last:"Vega"){
    id
    firstName
    lastName
    email
    orders {
      id
      product {
        id
        title
        desc
      }
      qty
      orderedOn
      status
    }
  }
}
```

Single HTTP Endpoint

GraphQL Java Engine

Data Fetcher

Data Fetcher

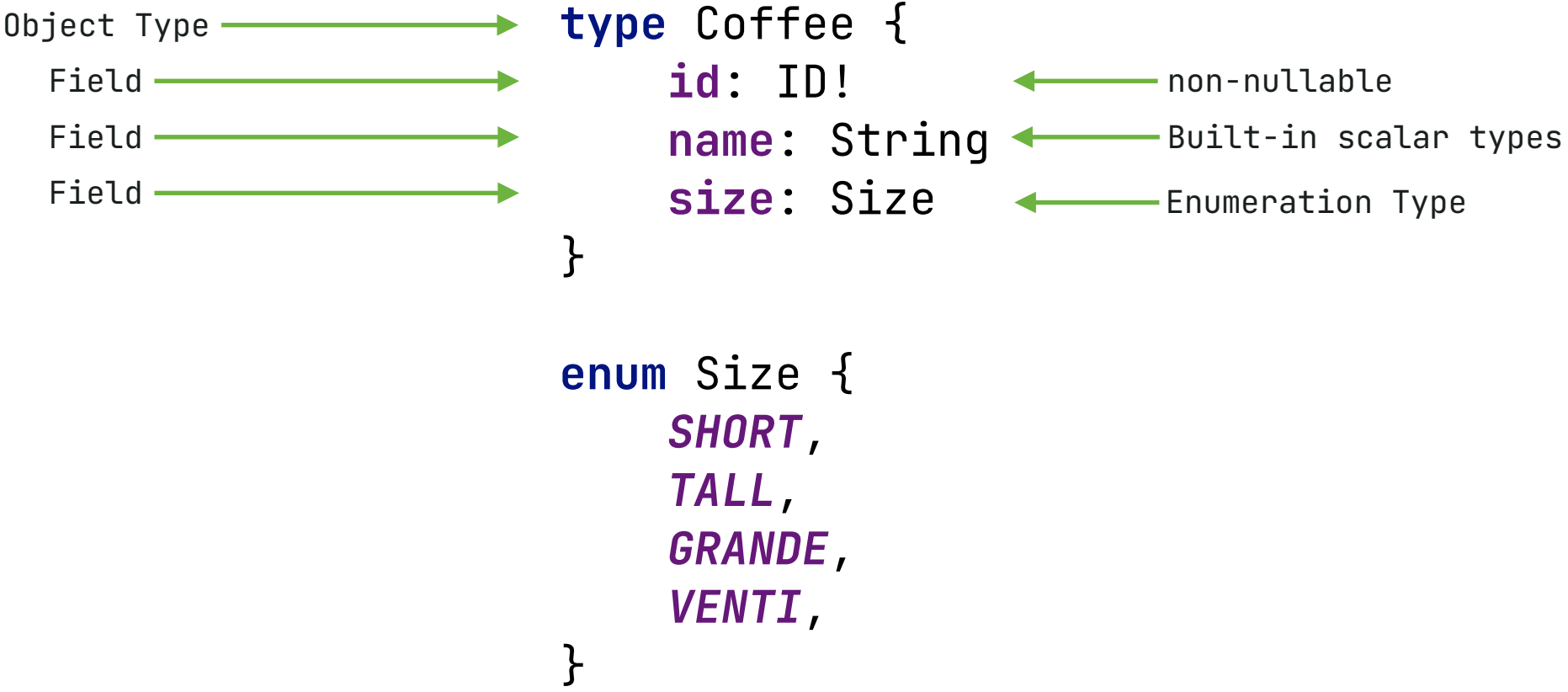
Data Fetcher

Data Fetcher

Data Fetcher

Application

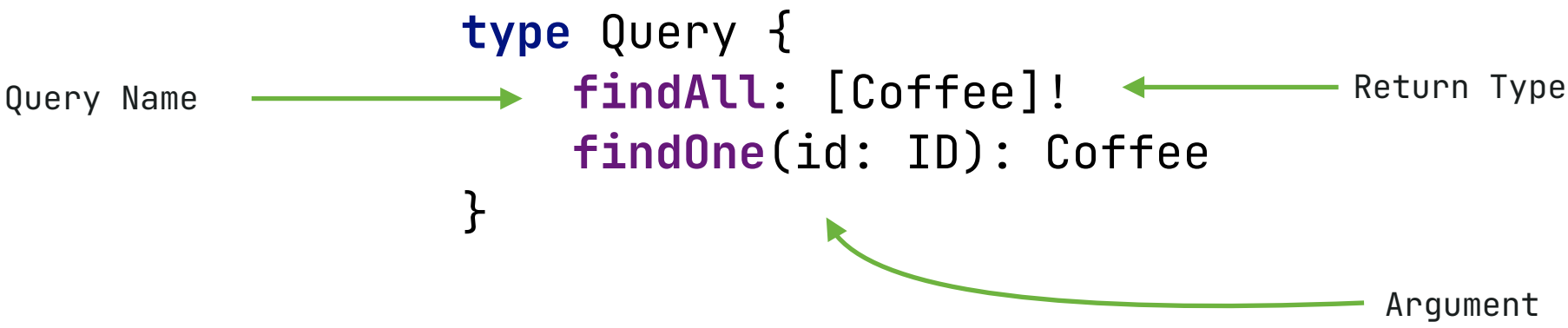
GraphQL Schema



src/main/resources/graphql/schema.graphqls

Operation Types

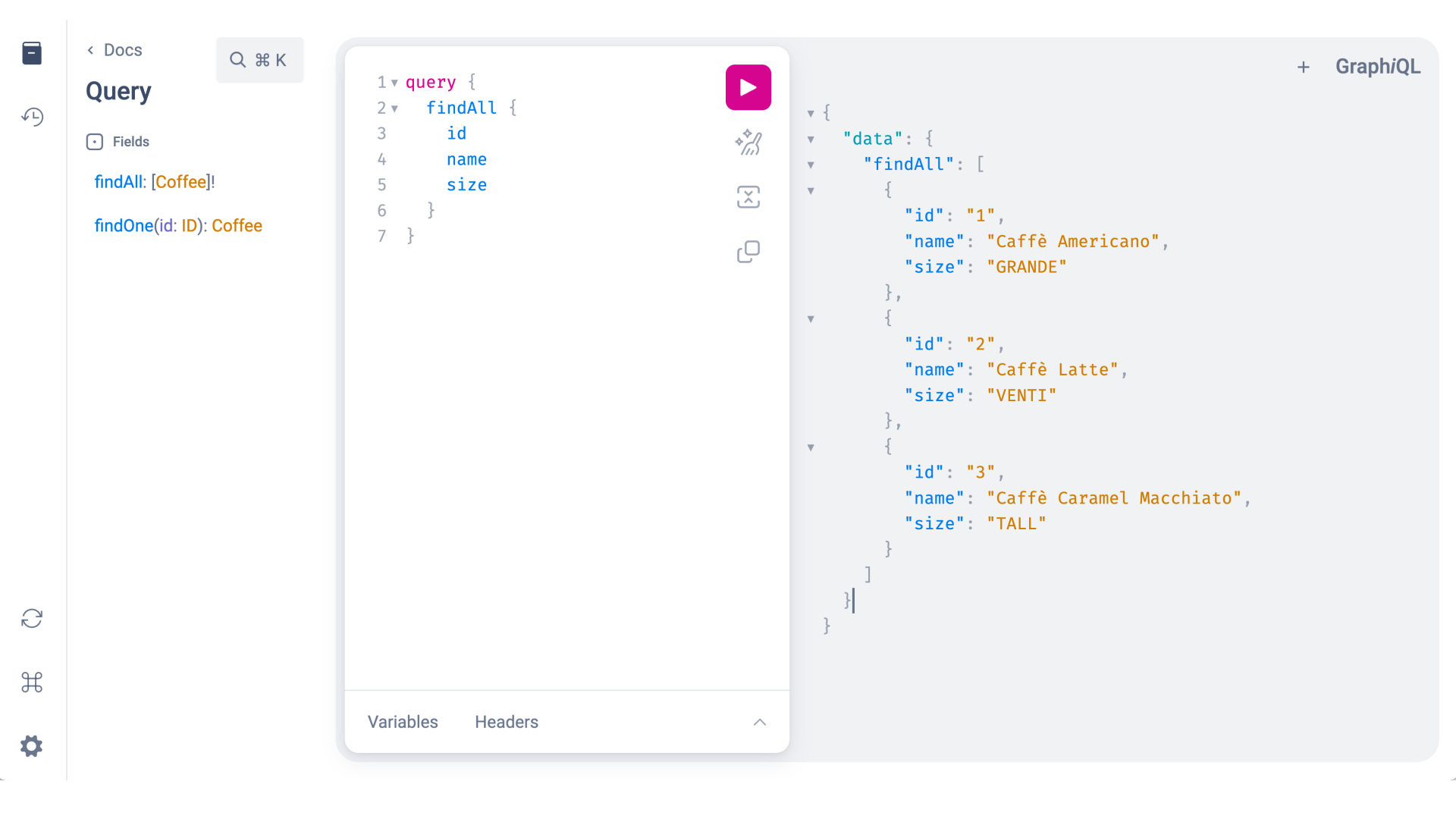
Most types in your schema will just be normal object types, but there are three types that are special within a schema:


type Query {
findAll: [Coffee]!
findOne(id: ID): Coffee
}
Query Name
Return Type
Argument

```java
@Controller
public class CoffeeController {

    private final CoffeeService coffeeService;

    public CoffeeController(CoffeeService coffeeService) {
        this.coffeeService = coffeeService;
    }

    public List<Coffee> list() {
        return coffeeService.findAll();
    }

    public Optional<Coffee> findOne(Integer id) {
        return coffeeService.findOne(id);
    }

}
```

```java
@Controller
public class CoffeeController {

    private final CoffeeService coffeeService;

    public CoffeeController(CoffeeService coffeeService) {
        this.coffeeService = coffeeService;
    }

    @SchemaMapping(typeName = "Query", value = "findAll")
    public List<Coffee> list() {
        return coffeeService.findAll();
    }

    public Optional<Coffee> findOne(Integer id) {
        return coffeeService.findOne(id);
    }

}
```

```java
@Controller
public class CoffeeController {

    private final CoffeeService coffeeService;

    public CoffeeController(CoffeeService coffeeService) {
        this.coffeeService = coffeeService;
    }

    @QueryMapping
    public List<Coffee> findAll() {
        return coffeeService.findAll();
    }

    @QueryMapping
    public Optional<Coffee> findOne(@Argument Integer id) {
        return coffeeService.findOne(id);
    }

}
```


Docs
⌘ K
Query
Fields
findAll: [Coffee]!
findOne(id: ID): Coffee
query {
findAll {
id
name
size
}
}
Variables
Headers
GraphiQL
{
"data": {
"findAll": [
{
"id": "1",
"name": "Caffè Americano",
"size": "GRANDE"
},
{
"id": "2",
"name": "Caffè Latte",
"size": "VENTI"
},
{
"id": "3",
"name": "Caffè Caramel Macchiato",
"size": "TALL"
}
]
}
}

Problem

I need to write some code that will call another service. This could be another service in your organization or an API you need to consume.

Solution

There are many solutions to make service to service calls in **Java** and **Spring**. In Java you have HttpClient, Apache HttpClient and OkHttp. In Spring you have access to RestTemplate and Web Client and in Spring Boot 3 you can use HTTP Interfaces.

Package org.springframework.web.client

Class RestTemplate

java.lang.Object
org.springframework.http.client.support.HttpAccessor
org.springframework.http.client.support.InterceptingHttpAccessor
org.springframework.web.client.RestTemplate

All Implemented Interfaces:

`RestOperations`

```
public class RestTemplate
extends InterceptingHttpAccessor
implements RestOperations
```

Synchronous client to perform HTTP requests, exposing a simple, template method API over underlying HTTP client libraries such as the JDK `HttpURLConnection`, Apache HttpComponents, and others. RestTemplate offers templates for common scenarios by HTTP method, in addit
generalized `exchange` and `execute` methods that support less frequent cases.

RestTemplate is typically used as a shared component. However, its configuration does not support concurrent modification, and as such its configuration is typically prepared on startup. If necessary, you can create multiple, differently configured RestTemplate instances on startup. Suc
may use the same underlying `ClientHttpRequestFactory` if they need to share HTTP client resources.

NOTE: As of 5.0 this class is in maintenance mode, with only minor requests for changes and bugs to be accepted going forward. Please, consider using the `org.springframework.web.reactive.client.WebClient` which has a more modern API and supports sync, async, and s
scenarios.

Since:

3.0

Author:

Arjen Poutsma, Brian Clozel, Roy Clarkson, Juergen Hoeller, Sam Brannen, Sebastien Deleuze

See Also:

`HttpMessageConverter`, `RequestCallback`, `ResponseExtractor`, `ResponseErrorHandler`

Field Summary

Fields inherited from class org.springframework.http.client.support.HttpAccessor

`logger`

Constructor Summary

Constructors

Constructor	Description
`RestTemplate()`	Create a new instance of the `RestTemplate` using default settings.

```java
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Bean
    RestTemplate restTemplate() {
        return new RestTemplateBuilder().build();
    }

}
```

```java
@RestController
@RequestMapping("/api/coffee")
public class CoffeeController {

    private final CoffeeService coffeeService;
    private final RestTemplate restTemplate;

    public CoffeeController(CoffeeService coffeeService, RestTemplate restTemplate) {
        this.coffeeService = coffeeService;
        this.restTemplate = restTemplate;
    }

    @GetMapping
    public List<Coffee> findAll() {
        return coffeeService.findAll();
    }

    @GetMapping("/random")
     public CoffeeImage random() {
       return new CoffeeImage(new Random().nextInt(100),
               restTemplate.getForObject("https://coffee.alexflipnote.dev/random.json", String.class));
     }

}
```

Package org.springframework.web.reactive.function.client

Interface WebClient

```
public interface WebClient
```

Non-blocking, reactive client to perform HTTP requests, exposing a fluent, reactive API over underlying HTTP client libraries such as Reactor Netty.

Use static factory methods `create()` or `create(String)`, or `builder()` to prepare an instance.

For examples with a response body see:

- `retrieve()`
- `exchangeToMono()`
- `exchangeToFlux()`

For examples with a request body see:

- `bodyValue(Object)`
- `body(Publisher,Class)`

Since:

5.0

Author:

Rossen Stoyanchev, Arjen Poutsma, Sebastien Deleuze, Brian Clozel

Nested Class Summary

Nested Classes

Modifier and Type	Interface	Description
static interface	**WebClient.Builder**	A mutable builder for creating a `WebClient`.
static interface	**WebClient.RequestBodySpec**	Contract for specifying request headers and body leading up to the exchange.
static interface	**WebClient.RequestBodyUriSpec**	Contract for specifying request headers, body and URI for a request.
static interface	**WebClient.RequestHeadersSpec<S extends WebClient.RequestHeadersSpec<S>>**	Contract for specifying request headers leading up to the exchange.
static interface	**WebClient.RequestHeadersUriSpec<S extends WebClient.RequestHeadersSpec<S>>**	Contract for specifying request headers and URI for a request.
static interface	**WebClient.ResponseSpec**	Contract for specifying response operations following the exchange.
static interface	**WebClient.UriSpec<S extends WebClient.RequestHeadersSpec<?>>**	Contract for specifying the URI for a request.

Method Summary

```java
@RestController
@RequestMapping("/api/coffee")
public class CoffeeController {

    WebClient client = WebClient.create("https://coffee.alexflipnote.dev");

    @GetMapping("/random")
    public Mono<Coffee> random() {
        return client.get()
                .uri("/random.json")
                .retrieve()
                .bodyToMono(Coffee.class);
    }
}
```

GraphQL Client

```java
@Service
public class CountryService {

    private HttpGraphQlClient graphQlClient;

    public CountryService() {
        WebClient client = WebClient.builder()
                .baseUrl("https://countries.trevorblades.com")
                .build();
        graphQlClient = HttpGraphQlClient.builder(client).build();
    }

}
```

```java
public Mono<List<Country>> getCountries() {
    //language=GraphQL
    String document = """
        query {
          countries {
            name
            emoji
            currency
            code
            capital
          }
        }
    """;

    Mono<List<Country>> countries = graphQlClient.document(document)
            .retrieve("countries")
            .toEntityList(Country.class);

    return countries;
}
```

```java
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Bean
    CommandLineRunner commandLineRunner(CountryService service,CountryRepository repository) {
        return args -> {
            Mono<List<Country>> countries = service.getCountries();
            countries.subscribe(response -> {
                repository.saveAll(response);
            });
        };
    }

}
```

```java
@Controller
public class CountryController {

    private final CountryRepository repository;

    public CountryController(CountryRepository repository) {
        this.repository = repository;
    }

    @QueryMapping
    public List<Country> findAllCountries() {
        return repository.findAll();
    }

}
```

HTTP Interfaces

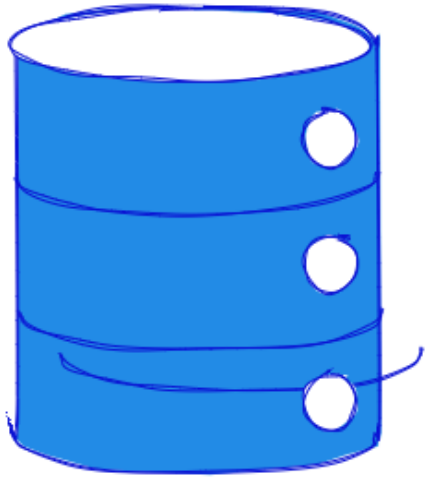
```java
package dev.danvega.http.client;

import dev.danvega.http.model.Coffee;
import org.springframework.web.service.annotation.GetExchange;

public interface CoffeeClient {

    @GetExchange("/random.json")
    Coffee random();

}
```

```java
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Bean
    CoffeeClient coffeeClient() {
        WebClient client = WebClient.builder()
                .baseUrl("https://coffee.alexflipnote.dev")
                .build();
        HttpServiceProxyFactory factory = HttpServiceProxyFactory
                .builder(WebClientAdapter.forClient(client))
                .build();
        return factory.createClient(CoffeeClient.class);
    }

}
```

```java
@RestController
@RequestMapping("/api/coffee")
public class CoffeeController {

    private final CoffeeClient coffeeClient;

    public CoffeeController(CoffeeClient coffeeClient) {
        this.coffeeClient = coffeeClient;
    }

    @GetMapping("/random")
    public Coffee random() {
        return coffeeClient.random();
    }

}
```

Working with Databases

Java + Databases



Problem

I need to create an application that can connect to a database.

Solution

Connecting to a database with Spring Boot is really easy. The quickest way to get up and running is by selecting an in-memory database.

spring initializr

Project

- (●) Maven Project
- ( ) Gradle Project

Language

- (●) Java
- ( ) Kotlin
- ( ) Groovy

Spring Boot

- ( ) 3.0.0 (SNAPSHOT)
- (●) 3.0.0 (M4)
- ( ) 2.7.3 (SNAPSHOT)
- ( ) 2.7.2
- ( ) 2.6.11 (SNAPSHOT)
- ( ) 2.6.10

Project Metadata

Group	dev.danvega
Artifact	h2-demo
Name	h2-demo
Description	Demo project for Spring Boot
Package name	dev.danvega.h2-demo
Packaging	(●) Jar ( ) War
Java	( ) 18 (●) 17 ( ) 11 ( ) 8

Dependencies ADD DEPENDENCIES... ⌘ + B

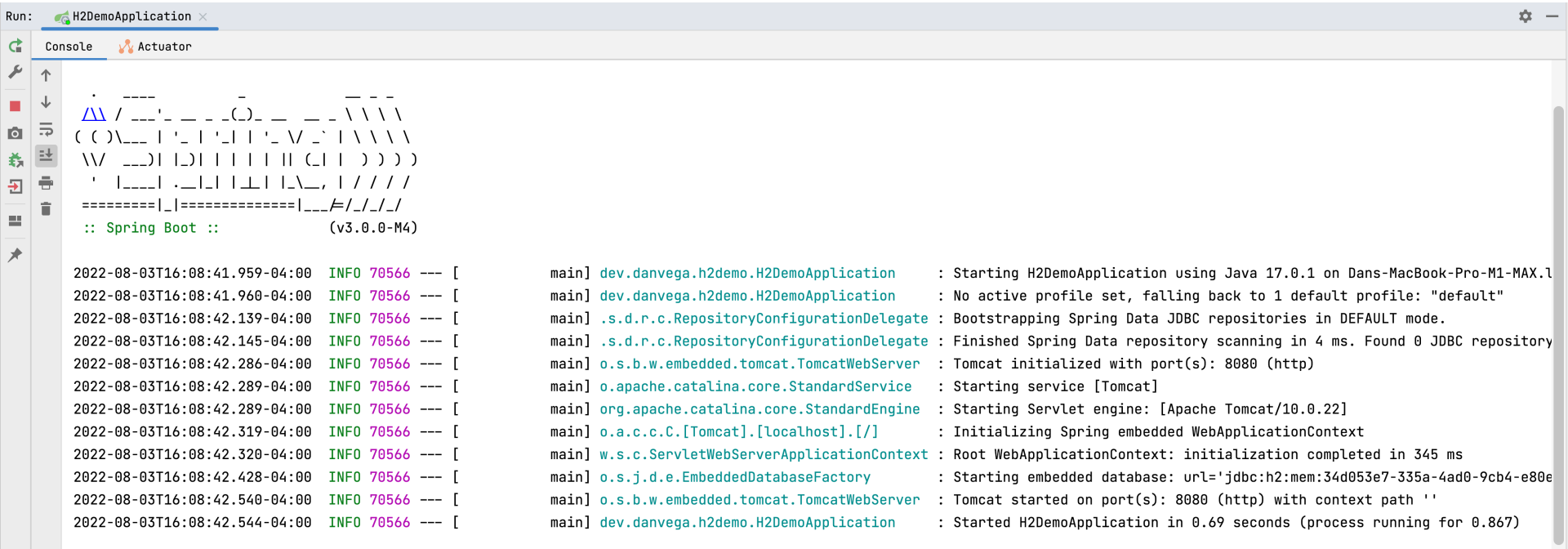
Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Data JDBC SQL
Persist data in SQL stores with plain JDBC using Spring Data.

H2 Database SQL
Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.

GENERATE ⌘ + ↵ EXPLORE CTRL + SPACE SHARE...

Starting embedded database:
url='jdbc:h2:mem:89060c30-14bd-4f81-8606-3c6f274c94bf;
DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=false',
username='sa'



```
spring.datasource.generate-unique-name=false
spring.datasource.name=coffee
spring.h2.console.enabled=true
```

application.properties

Run Console Actuator

```
/Users/vega/.sdkman/candidates/java/17.0.5-oracle/bin/java ...

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v3.0.4)

2023-03-28T09:23:46.138-04:00  INFO 96795 --- [           main] dev.danvega.h2demo.Application           : Starting Application using Java 17.0.5 with PID 96795 (
2023-03-28T09:23:46.140-04:00  INFO 96795 --- [           main] dev.danvega.h2demo.Application           : No active profile set, falling back to 1 default profile
2023-03-28T09:23:46.351-04:00  INFO 96795 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JDBC repositories in DEFAULT m
2023-03-28T09:23:46.358-04:00  INFO 96795 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 5 ms. Found
2023-03-28T09:23:46.529-04:00  INFO 96795 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2023-03-28T09:23:46.532-04:00  INFO 96795 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2023-03-28T09:23:46.532-04:00  INFO 96795 --- [           main] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache Tomcat/10.1.5]
2023-03-28T09:23:46.566-04:00  INFO 96795 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2023-03-28T09:23:46.566-04:00  INFO 96795 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in
2023-03-28T09:23:46.575-04:00  INFO 96795 --- [           main] o.s.j.d.e.EmbeddedDatabaseFactory        : Starting embedded database: url='jdbc:h2:mem:coffee;DB_C
2023-03-28T09:23:46.649-04:00  INFO 96795 --- [           main] o.s.b.a.h2.H2ConsoleAutoConfiguration    : H2 console available at '/h2-console'. Database availabl
2023-03-28T09:23:46.854-04:00  INFO 96795 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path
2023-03-28T09:23:46.858-04:00  INFO 96795 --- [           main] dev.danvega.h2demo.Application           : Started Application in 0.865 seconds (process running fo
```

Starting embedded database: url='jdbc:h2:mem:coffee;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=false', username='sa'

H2 console available at '/h2-console'. Database available at 'jdbc:h2:mem:coffee'

English Preferences Tools Help

Login

Saved Settings: Generic H2 (Embedded)

Setting Name: Generic H2 (Embedded) Save Remove

Driver Class: org.h2.Driver

JDBC URL: jdbc:h2:mem:coffee

User Name: sa

Password:

Connect Test Connection

Test successful

jdbc:h2:mem:coffee
INFORMATION_SCHEMA
Users
H2 2.1.214 (2022-06-13)

Run | Run Selected | Auto complete | Clear | SQL statement:

```
select * from coffee
```

Important Commands

		Displays this Help Page
		Shows the Command History
	Ctrl+Enter	Executes the current SQL statement
	Shift+Enter	Executes the SQL statement defined by the text selection
	Ctrl+Space	Auto complete
		Disconnects from the database

Sample SQL Script

Delete the table if it exists	DROP TABLE IF EXISTS TEST;
Create a new table with ID and NAME columns	CREATE TABLE TEST(ID INT PRIMARY KEY, NAME VARCHAR(255));
Add a new row	INSERT INTO TEST VALUES(1, 'Hello');
Add another row	INSERT INTO TEST VALUES(2, 'World');
Query the table	SELECT * FROM TEST ORDER BY ID;

Problem

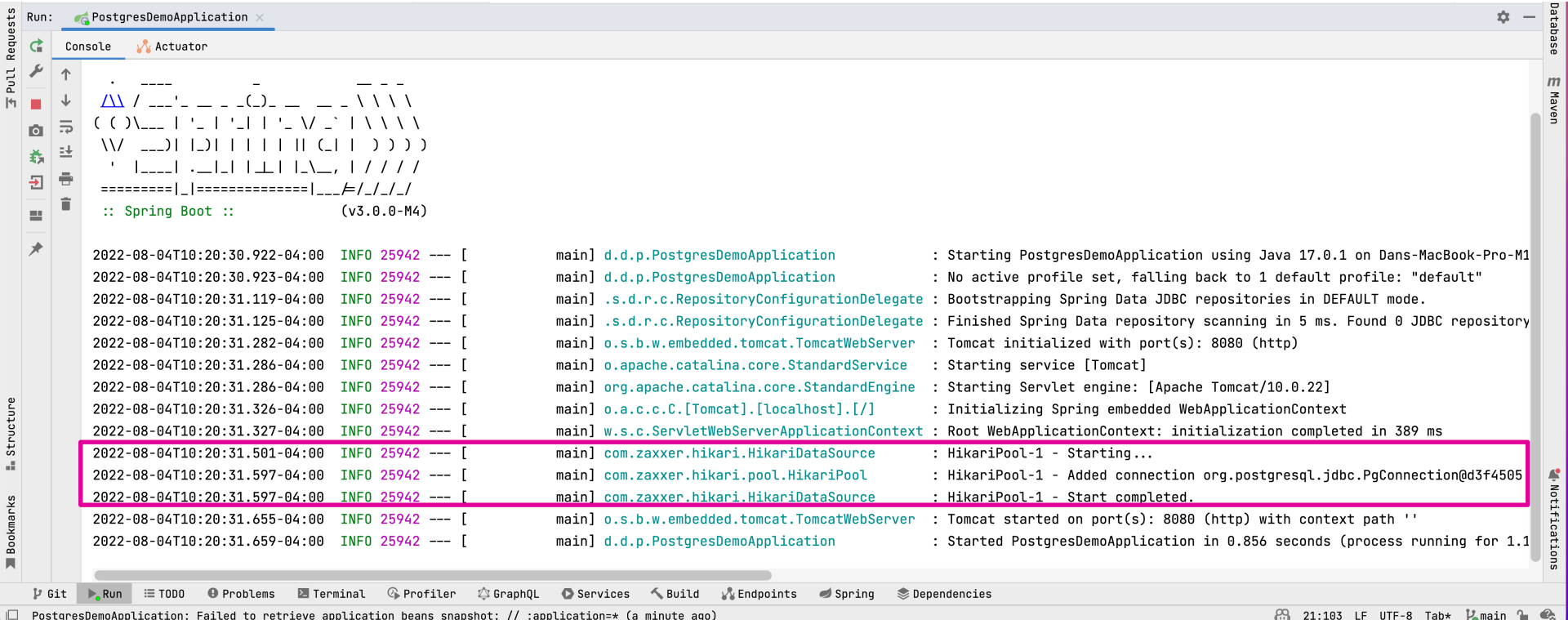
That was a great example but how can I connect to a production ready database?

Solution

Connecting to a database like Postgres is just a few more lines of configuration.

```yaml
version: '3.8'

services:
  db:
    image: postgres:alpine
    ports:
      - "5432:5432"
    environment:
      POSTGRES_PASSWORD: password
      POSTGRES_DB: coffee
```

```
spring.datasource.url=jdbc:postgresql://localhost:5432/coffee
spring.datasource.username=postgres
spring.datasource.password=password
```


Run: PostgresDemoApplication
Console
Actuator
:: Spring Boot :: (v3.0.0-M4)
2022-08-04T10:20:30.922-04:00 INFO 25942 --- [main] d.d.p.PostgresDemoApplication : Starting PostgresDemoApplication using Java 17.0.1 on Dans-MacBook-Pro-M1
2022-08-04T10:20:30.923-04:00 INFO 25942 --- [main] d.d.p.PostgresDemoApplication : No active profile set, falling back to 1 default profile: "default"
2022-08-04T10:20:31.119-04:00 INFO 25942 --- [main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JDBC repositories in DEFAULT mode.
2022-08-04T10:20:31.125-04:00 INFO 25942 --- [main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 5 ms. Found 0 JDBC repository
2022-08-04T10:20:31.282-04:00 INFO 25942 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2022-08-04T10:20:31.286-04:00 INFO 25942 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2022-08-04T10:20:31.286-04:00 INFO 25942 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.0.22]
2022-08-04T10:20:31.326-04:00 INFO 25942 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2022-08-04T10:20:31.327-04:00 INFO 25942 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 389 ms
2022-08-04T10:20:31.501-04:00 INFO 25942 --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2022-08-04T10:20:31.597-04:00 INFO 25942 --- [main] com.zaxxer.hikari.pool.HikariPool : HikariPool-1 - Added connection org.postgresql.jdbc.PgConnection@d3f4505
2022-08-04T10:20:31.597-04:00 INFO 25942 --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2022-08-04T10:20:31.655-04:00 INFO 25942 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2022-08-04T10:20:31.659-04:00 INFO 25942 --- [main] d.d.p.PostgresDemoApplication : Started PostgresDemoApplication in 0.856 seconds (process running for 1.1
Git
Run
TODO
Problems
Terminal
Profiler
GraphQL
Services
Build
Endpoints
Spring
Dependencies
PostgresDemoApplication: Failed to retrieve application beans snapshot: // :application=* (a minute ago)
21:103 LF UTF-8 Tab* main

Problem

Now that I am connected to a database how can I run a DDL script to create tables, columns and data?

Solution

You can create a SQL script by convention in *src/main/resources/schema.sql*.

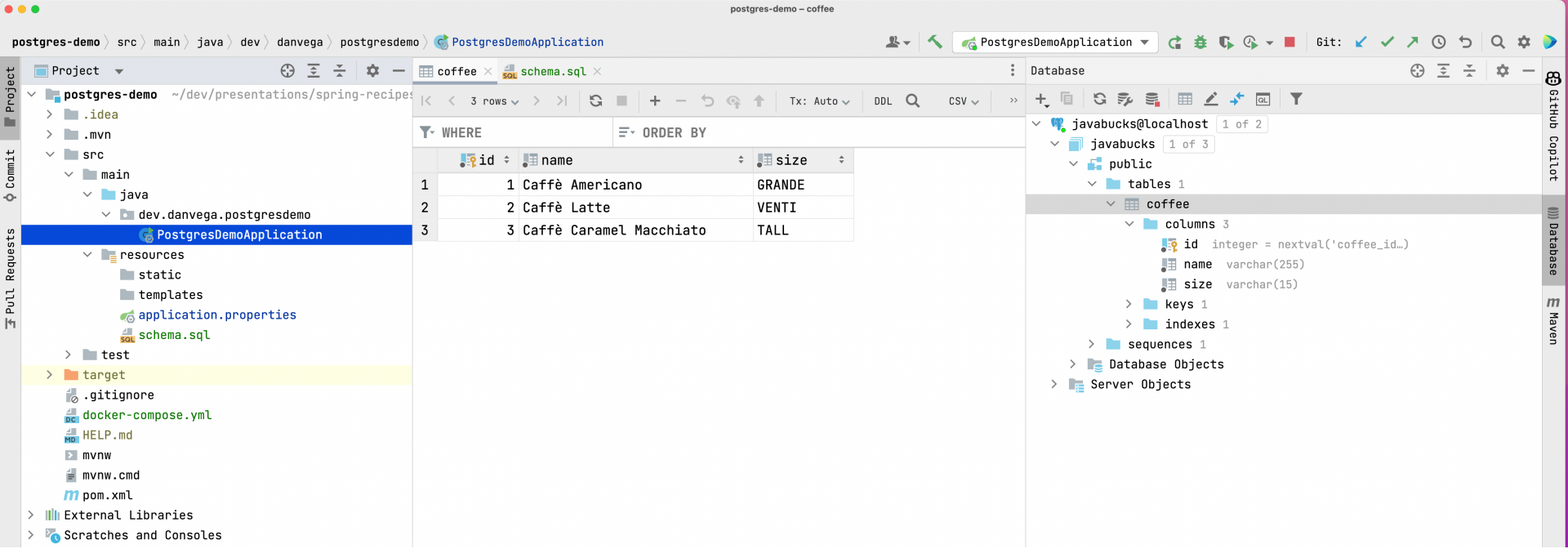
```sql
CREATE TABLE coffee(
    id SERIAL NOT NULL PRIMARY KEY,
    name varchar(255) NOT NULL,
    size varchar(15) NOT NULL
);
```

src/main/resources/schema.sql

```
spring.datasource.url=jdbc:postgresql://localhost:5432/javabucks
spring.datasource.username=postgres
spring.datasource.password=password

spring.sql.init.mode=always
```

```sql
CREATE TABLE coffee(
    id SERIAL NOT NULL PRIMARY KEY,
    name varchar(255) NOT NULL,
    size varchar(15) NOT NULL
);
```

```sql
INSERT INTO coffee(name,size) VALUES('Caffè Americano','GRANDE');
INSERT INTO coffee(name,size) VALUES('Caffè Latte','VENTI');
INSERT INTO coffee(name,size) VALUES('Caffè Caramel Macchiato','TALL');
```


postgres-demo – coffee
postgres-demo 〉 src 〉 main 〉 java 〉 dev 〉 danvega 〉 postgresdemo 〉 PostgresDemoApplication
PostgresDemoApplication
Git:
Project
postgres-demo ~/dev/presentations/spring-recipes
.idea
.mvn
src
main
java
dev.danvega.postgresdemo
PostgresDemoApplication
resources
static
templates
application.properties
schema.sql
test
target
.gitignore
docker-compose.yml
HELP.md
mvnw
mvnw.cmd
pom.xml
External Libraries
Scratches and Consoles
coffee
schema.sql
3 rows
Tx: Auto
DDL
CSV
WHERE
ORDER BY
id
name
size
1 1 Caffè Americano GRANDE
2 2 Caffè Latte VENTI
3 3 Caffè Caramel Macchiato TALL
Database
javabucks@localhost 1 of 2
javabucks 1 of 3
public
tables 1
coffee
columns 3
id integer = nextval('coffee_id…)
name varchar(255)
size varchar(15)
keys 1
indexes 1
sequences 1
Database Objects
Server Objects
GitHub Copilot
Database
Maven
Commit
Pull Requests

Problem

You want to be able to insert data programmatically using Java & Spring.

Solution

You can use JDBC Template or Spring Data to persist data. The ***ApplicationRunner*** and ***CommandLineRunner*** interfaces in Spring are a great way to bootstrap data.

```java
@SpringBootApplication
public class PostgresDemoApplication {


    public static void main(String[] args) {
        SpringApplication.run(PostgresDemoApplication.class, args);
    }

    @Bean
    CommandLineRunner runner(CoffeeDAO dao) {
        return args -> {
            dao.create(new Coffee(null,"Caffè Americano", Size.GRANDE));
            dao.create(new Coffee(null,"Caffè Latte", Size.VENTI));
            dao.create(new Coffee(null,"Caffè Caramel Macchiato", Size.TALL));
        };
    }

}
```

```java
@Component
public class DatabaseLoader implements CommandLineRunner {

    private static final Logger LOG = LoggerFactory.getLogger(DatabaseLoader.class);

    private final CoffeeRepository repository;

    public DatabaseLoader(CoffeeRepository coffeeRepository) {
        this.repository = coffeeRepository;
    }

    @Override
    public void run(String... args) throws Exception {
        List<Coffee> coffees = List.of(new Coffee(null, "Caffè Americano", Size.GRANDE),
                new Coffee(null, "Caffè Latte", Size.VENTI),
                new Coffee(null, "Caffè Caramel Macchiato", Size.TALL));
        repository.saveAll(coffees);
    }

}
```

Spring Cloud

Problem

You need to monitor various metrics, information or the health information of a service or application.

Solution

Spring Boot Actuator allows you to monitor and interact with your application via a number of pre built endpoints. Developers can also create custom endpoints.

Back to index

1. Enabling Production-ready Features
2. Endpoints
   2.1. Enabling Endpoints
   2.2. Exposing Endpoints
   2.3. Security
   2.4. Configuring Endpoints
   2.5. Hypermedia for Actuator Web Endpoints
   2.6. CORS Support
   2.7. Implementing Custom Endpoints
   2.8. Health Information
   2.9. Kubernetes Probes
   2.10. Application Information
3. Monitoring and Management Over HTTP
4. Monitoring and Management over JMX
5. Observability
6. Loggers
7. Metrics
8. Tracing
9. Auditing
10. Recording HTTP Exchanges
11. Process Monitoring
12. Cloud Foundry Support
13. What to Read Next

2. Endpoints

Actuator endpoints let you monitor and interact with your application. Spring Boot includes a number of built-in endpoints and lets you add your own. For example, the `health` endpoint provides basic application health information.

You can enable or disable each individual endpoint and expose them (make them remotely accessible) over HTTP or JMX. An endpoint is considered to be available when it is both enabled and exposed. The built-in endpoints are auto-configured only when they are available. Most applications choose exposure over HTTP, where the ID of the endpoint and a prefix of `/actuator` is mapped to a URL. For example, by default, the `health` endpoint is mapped to `/actuator/health`.

Tip

To learn more about the Actuator's endpoints and their request and response formats, see the separate API documentation (HTML or PDF).

The following technology-agnostic endpoints are available:

ID	Description
`auditevents`	Exposes audit events information for the current application. Requires an `AuditEventRepository` bean.
`beans`	Displays a complete list of all the Spring beans in your application.
`caches`	Exposes available caches.
`conditions`	Shows the conditions that were evaluated on configuration and auto-configuration classes and the reasons why they did or did not match.
`configprops`	Displays a collated list of all `@ConfigurationProperties`.
`env`	Exposes properties from Spring's `ConfigurableEnvironment`.
`flyway`	Shows any Flyway database migrations that have been applied. Requires one or more `Flyway` beans.
`health`	Shows application health information.
`httpexchanges`	Displays HTTP exchange information (by default, the last 100 HTTP request-response exchanges). Requires an `HttpExchangeRepository` bean.
`info`	Displays arbitrary application info.
`integrationgraph`	Shows the Spring Integration graph. Requires a dependency on `spring-integration-core`.

```xml
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
</dependencies>
```

```
management:
  endpoint:
    shutdown:
      enabled: true
```

Problem

Your application needs to degrade gracefully when services fail.

Solution

Spring Cloud Circuit Breaker provides an abstraction over various implementations of the circuit breaker pattern allowing your application to avoid cascading failures.

Spring Cloud Circuit Breaker 3.0.1

Introduction

Spring Cloud Circuit breaker provides an abstraction across different circuit breaker implementations. It provides a consistent API to use in your applications allowing you the developer to choose the circuit breaker implementation that best fits your needs for your app.

Supported Implementations

- Resilience4J
- Spring Retry

Core Concepts

To create a circuit breaker in your code you can use the `CircuitBreakerFactory` API. When you include a Spring Cloud Circuit Breaker starter on your classpath a bean implementing this API will automatically be created for you. A very simple example of using this API is given below

COPY

```
@Service
public static class DemoControllerService {
        private RestTemplate rest;
        private CircuitBreakerFactory cbFactory;

        public DemoControllerService(RestTemplate rest, CircuitBreakerFactory cbFactory) {
                this.rest = rest;
                this.cbFactory = cbFactory;
        }
```

```xml
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>
</dependency>
```

```
CircuitBreakerConfig circuitBreakerConfig = CircuitBreakerConfig.custom()
        .failureRateThreshold(5)
        .waitDurationInOpenState(Duration.ofMillis(300))
        .slidingWindowSize(2)
        .build();
```

```
WaffleController(MacronutrientsProvider macronutrientsProvider, CircuitBreakerFactory circuitBreakerFactory) {
    this.macronutrientsProvider = macronutrientsProvider;
    this.circuitBreaker = circuitBreakerFactory.create("waffles");
}

@GetMapping("/create")
Waffle create() {
    return new Waffle(circuitBreaker.run(macronutrientsProvider::protein));
}
```

Problem

To meet your application's scaling needs you've adopted an event driven approach but you don't want to tie yourself to one particular messaging broker.

Solution

Spring Cloud Stream provides binders for multiple message brokers from RabbitMQ to Apache Kafka to various cloud based pub/sub implementations.

Spring Cloud Stream 4.0.2

Spring Cloud Stream is a framework for building highly scalable event-driven microservices connected with shared messaging systems.

The framework provides a flexible programming model built on already established and familiar Spring idioms and best practices, including support for persistent pub/sub semantics, consumer groups, and stateful partitions.

Binder Implementations

Spring Cloud Stream supports a variety of binder implementations and the following table includes the link to the GitHub projects.

- RabbitMQ
- Apache Kafka
- Kafka Streams
- Amazon Kinesis
- Google PubSub *(partner maintained)*
- Solace PubSub+ *(partner maintained)*
- Azure Event Hubs *(partner maintained)*
- Azure Service Bus *(partner maintained)*
- AWS SQS *(partner maintained)*
- AWS SNS *(partner maintained)*
- Apache RocketMQ *(partner maintained)*

The core building blocks of Spring Cloud Stream are:

- **Destination Binders**: Components responsible to provide integration with the external messaging systems.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
</dependency>
```

Problem

Your service is used by a variety of consumers and you cannot introduce breaking changes to downstream systems.

Solution

Spring Cloud Contract simplifies consumer driven contracts bringing test driven development to your application architecture.

Spring Cloud Contract 4.0.2

Spring Cloud Contract is an umbrella project holding solutions that help users in successfully implementing the Consumer Driven Contracts approach. Currently Spring Cloud Contract consists of the Spring Cloud Contract Verifier project.

Spring Cloud Contract Verifier is a tool that enables Consumer Driven Contract (CDC) development of JVM-based applications. It is shipped with Contract Definition Language (DSL) written in Groovy or YAML. Contract definitions are used to produce following resources:

- by default JSON stub definitions to be used by WireMock (HTTP Server Stub) when doing integration testing on the client code (client tests). Test code must still be written by hand, test data is produced by Spring Cloud Contract Verifier.
- Messaging routes if you're using one. We're integrating with Spring Integration, Spring Cloud Stream and Apache Camel. You can however set your own integrations if you want to.
- Acceptance tests (by default in JUnit or Spock) used to verify if server-side implementation of the API is compliant with the contract (server tests). Full test is generated by Spring Cloud Contract Verifier.

Spring Cloud Contract Verifier moves TDD to the level of software architecture.

To see how Spring Cloud Contract supports other languages just check out this blog post.

Features

When trying to test an application that communicates with other services then 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. Let's focus on the latter. Deploy all microservices and perform end to end tests

Advantages:

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-contract-verifier</artifactId>
    <scope>test</scope>
</dependency>
```

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-contract-stub-runner</artifactId>
    <scope>test</scope>
</dependency>
```

Security

Problem

In the context of upgrading to Spring Boot 3 and Spring Security 6, what are the key changes to be aware of, and how should one go about configuring Spring Security?

Solution

Spring Security 6 introduces a new component based configuration and deprecated some APIs.

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v3.0.5)

2023-03-28T11:08:11.306-04:00  INFO 11013 --- [           main] dev.danvega.javabucks.Application        : Starting Application using Java 17.0.5 with PID 11013 (/Users/vega/dev/
2023-03-28T11:08:11.308-04:00  INFO 11013 --- [           main] dev.danvega.javabucks.Application        : No active profile set, falling back to 1 default profile: "default"
2023-03-28T11:08:11.557-04:00  INFO 11013 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JDBC repositories in DEFAULT mode.
2023-03-28T11:08:11.578-04:00  INFO 11013 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 18 ms. Found 1 JDBC reposit
2023-03-28T11:08:11.777-04:00  INFO 11013 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2023-03-28T11:08:11.782-04:00  INFO 11013 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2023-03-28T11:08:11.782-04:00  INFO 11013 --- [           main] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache Tomcat/10.1.7]
2023-03-28T11:08:11.818-04:00  INFO 11013 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2023-03-28T11:08:11.819-04:00  INFO 11013 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 491 ms
2023-03-28T11:08:11.833-04:00  INFO 11013 --- [           main] com.zaxxer.hikari.HikariDataSource       : coffee - Starting...
2023-03-28T11:08:11.930-04:00  INFO 11013 --- [           main] com.zaxxer.hikari.pool.HikariPool        : coffee - Added connection conn0: url=jdbc:h2:mem:coffee user=SA
2023-03-28T11:08:11.930-04:00  INFO 11013 --- [           main] com.zaxxer.hikari.HikariDataSource       : coffee - Start completed.
2023-03-28T11:08:11.935-04:00  INFO 11013 --- [           main] o.s.b.a.h2.H2ConsoleAutoConfiguration    : H2 console available at '/h2-console'. Database available at 'jdbc:h2:m
2023-03-28T11:08:12.193-04:00  WARN 11013 --- [           main] .s.s.UserDetailsServiceAutoConfiguration :

Using generated security password: 7400eece-d960-4587-90fb-419cde46f6e3

This generated password is for development use only. Your security configuration must be updated before running your application in production.

2023-03-28T11:08:12.245-04:00  INFO 11013 --- [           main] o.s.s.web.DefaultSecurityFilterChain     : Will secure any request with [org.springframework.security.web.session.
2023-03-28T11:08:12.267-04:00  INFO 11013 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2023-03-28T11:08:12.271-04:00  INFO 11013 --- [           main] dev.danvega.javabucks.Application        : Started Application in 1.132 seconds (process running for 1.414)
2023-03-28T11:08:38.604-04:00  INFO 11013 --- [nio-8080-exec-2] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
2023-03-28T11:08:38.604-04:00  INFO 11013 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
2023-03-28T11:08:38.605-04:00  INFO 11013 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet        : Completed initialization in 1 ms
```

```java
@Configuration
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests((authz) -> authz
                .anyRequest().authenticated()
            )
            .httpBasic(withDefaults());
    }

}
```

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http.build();
    }

}
```

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http.build();
    }

    @Bean
    UserDetailsService userDetailsService() {
        var user = User.withDefaultPasswordEncoder()
                .username("user")
                .password("password")
                .roles("USER")
                .build();

        return new InMemoryUserDetailsManager(user);
    }

}
```

```java
@Bean
SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    return http
            .authorizeHttpRequests( auth -> {
                auth.requestMatchers("/").permitAll();
                auth.requestMatchers("/api/coffee/**").hasRole("ADMIN");
                auth.anyRequest().authenticated();
            })
            .csrf(AbstractHttpConfigurer::disable)
            .httpBasic(Customizer.withDefaults())
            .build();
}
```

```java
@Bean
@Order(1)
SecurityFilterChain h2ConsoleSecurityFilterChain(HttpSecurity http) throws Exception {
    return http
            .securityMatcher(AntPathRequestMatcher.antMatcher("/h2-console/**"))
            .authorizeHttpRequests( auth -> {
                auth.requestMatchers(AntPathRequestMatcher.antMatcher("/h2-console/**")).permitAll();
            })
            .csrf(csrf -> csrf.ignoringRequestMatchers(AntPathRequestMatcher.antMatcher("/h2-console/**")))
            .headers(headers -> headers.frameOptions().disable())
            .build();
}

@Bean
SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    return http
            .authorizeHttpRequests( auth -> {
                auth.requestMatchers("/").permitAll();
                auth.requestMatchers("/api/coffee/**").hasRole("ADMIN");
                auth.anyRequest().authenticated();
            })
            .csrf(AbstractHttpConfigurer::disable)
            .httpBasic(Customizer.withDefaults())
            .build();
}
```

Problem

You would like to secure your REST APIs using JSON Web Tokens (JWT) but you aren't sure where to begin.

Solution

Spring Security provides built-in support for safeguarding endpoints using OAuth2 Resource Server, which includes the ability to utilize JWT as OAuth2 Bearer Tokens.

dan vega HOME BLOG COURSES NEWSLETTER SPEAKING Search (Press '/' to focus)

SPRING BOOT JWT - HOW TO SECURE YOUR REST APIS WITH SPRING SECURITY AND JSON WEB TOKENS

Published On: September 06, 2022 • 15 min read

The video tutorial for this blog post can be found above or you can click here to watch it on YouTube.

If you perform a quick search on how to secure REST APIs in Spring Boot using JSON Web Tokens you will find a lot of the same results. These results contain a method that involves writing a custom filter chain and pulling in a 3rd party library for encoding and decoding JWTs.

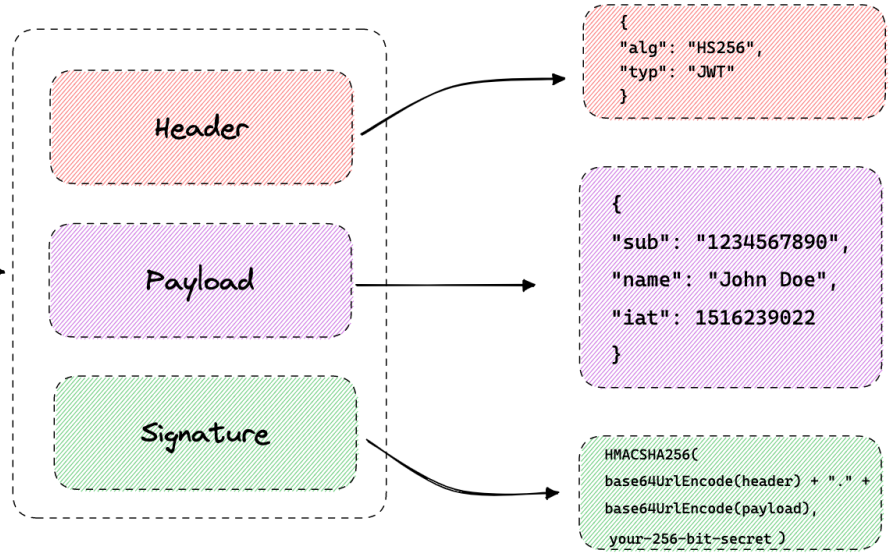
After staring at these convoluted and confusing tutorials I said there has to be an easier way to do this. I did what anyone with direct access to the Spring Security team would do, I asked them for help. They informed me that indeed Spring Security has built-in support for JWTs using oAuth2 Resource Server.

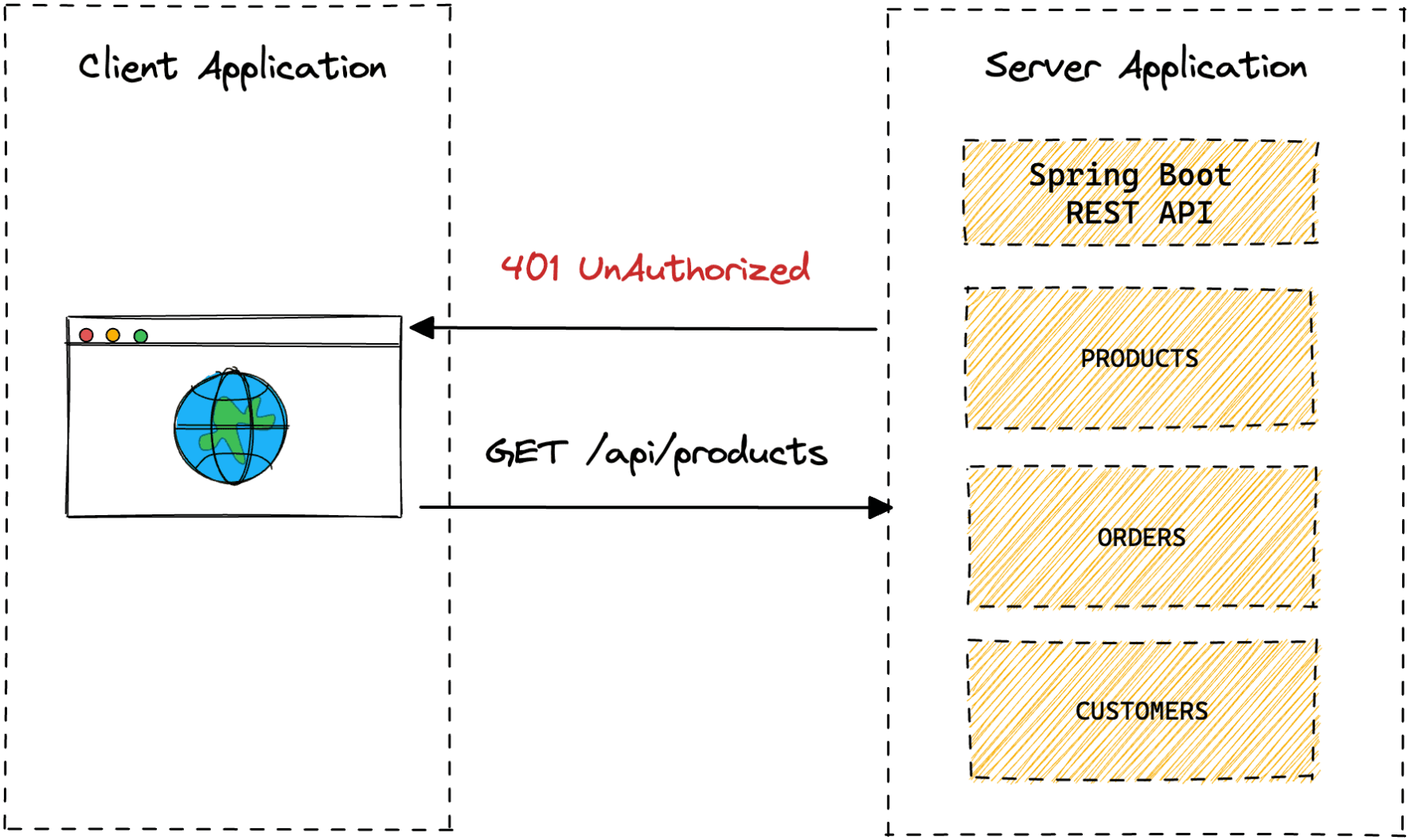
In this tutorial, you are going to learn how to secure your APIs using JSON Web Tokens (JWT) with Spring Security. I'm not saying this approach is easy by any stretch but for me, it made a lot more sense than the alternatives.

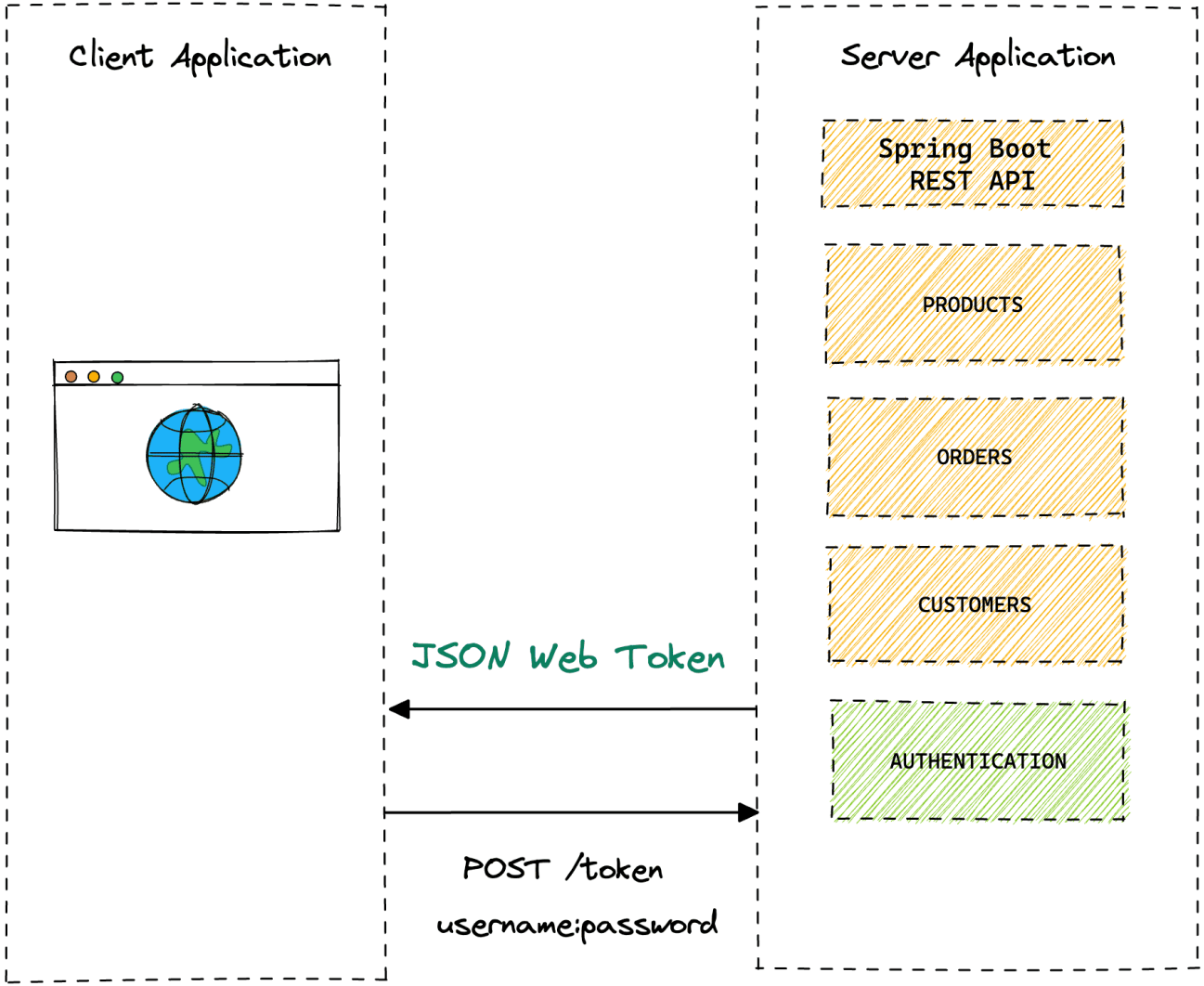
Github Repository

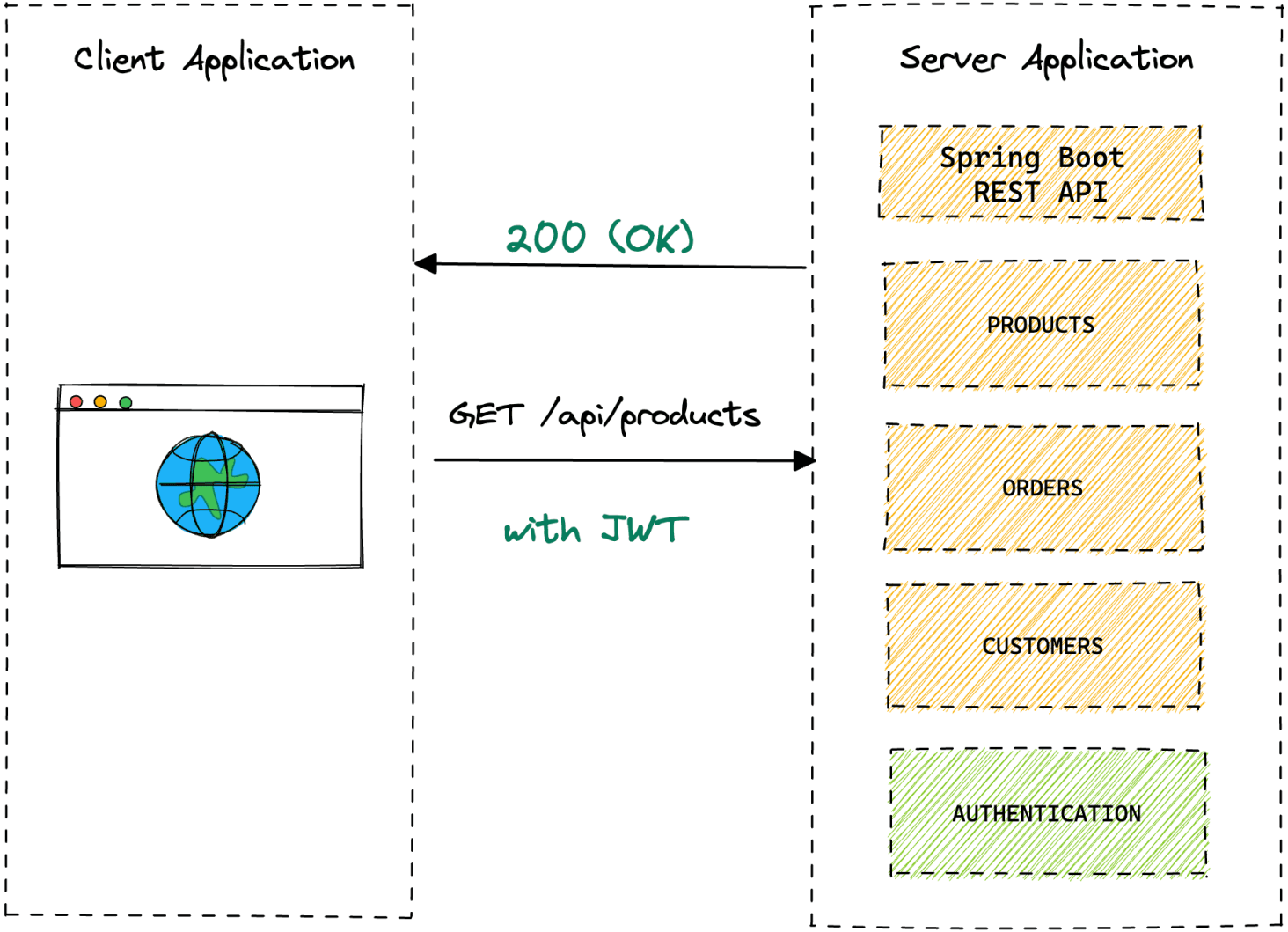
JSON Web Token (JWT)

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6
IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.
SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_ad
Qssw5c




Client Application
Server Application
Spring Boot
REST API
401 UnAuthorized
PRODUCTS
GET /api/products
ORDERS
CUSTOMERS


Client Application
Server Application
Spring Boot
REST API
PRODUCTS
ORDERS
CUSTOMERS
AUTHENTICATION
JSON Web Token
POST /token
username:password


Client Application
Server Application
Spring Boot
REST API
200 (OK)
PRODUCTS
GET /api/products
ORDERS
with JWT
CUSTOMERS
AUTHENTICATION

```java
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    return http
            .csrf(AbstractHttpConfigurer::disable)
            .authorizeHttpRequests( auth -> auth
                    .requestMatchers("/api/auth/token").hasRole("USER")
                    .anyRequest().hasAuthority("SCOPE_READ")
            )
            .sessionManagement(s -> s.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .oauth2ResourceServer(OAuth2ResourceServerConfigurer::jwt)
            .httpBasic(withDefaults())
            .build();
}
```

```java
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    return http
            .csrf(AbstractHttpConfigurer::disable)
            .authorizeHttpRequests( auth -> auth
                    .requestMatchers("/api/auth/token").hasRole("USER")
                    .anyRequest().hasAuthority("SCOPE_READ")
            )
            .sessionManagement(s -> s.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .oauth2ResourceServer(OAuth2ResourceServerConfigurer::jwt)
            .httpBasic(withDefaults())
            .build();
}
```

```java
@Bean
JwtEncoder jwtEncoder() {
    return new NimbusJwtEncoder(new ImmutableSecret<>(jwtKey.getBytes()));
}

@Bean
public JwtDecoder jwtDecoder() {
    byte[] bytes = jwtKey.getBytes();
    SecretKeySpec originalKey = new SecretKeySpec(bytes, 0, bytes.length,"RSA");
    return NimbusJwtDecoder.withSecretKey(originalKey).macAlgorithm(MacAlgorithm.HS512).build();
}
```

```java
private final JwtEncoder encoder;

public TokenService(JwtEncoder encoder) {
    this.encoder = encoder;
}

public String generateToken(Authentication authentication) {
    Instant now = Instant.now();
    String scope = authentication.getAuthorities().stream()
            .map(GrantedAuthority::getAuthority)
            .filter(authority -> !authority.startsWith("ROLE"))
            .collect(Collectors.joining(" "));
    JwtClaimsSet claims = JwtClaimsSet.builder()
            .issuer("self")
            .issuedAt(now)
            .expiresAt(now.plus(1, ChronoUnit.HOURS))
            .subject(authentication.getName())
            .claim("scope", scope)
            .build();
    var encoderParameters = JwtEncoderParameters.from(JwsHeader.with(MacAlgorithm.HS512).build(),
    return this.encoder.encode(encoderParameters).getTokenValue();
}
```

Spring Authorization Server Reference

Joe Grandja, Steve Riesenberg

Version 1.0.1

Version 1.0.1
Last updated 2023-02-21 17:26:45 UTC

Testing

Problem

As the size and complexity of my application increases my tests are taking longer and longer to run. How can I write tests that run faster?

Solution

A slice test allows you to focus on just a particular “slice” (web, db, etc…) of your application.

Test slice
@DataCassandraTest
@DataCouchbaseTest
@DataElasticsearchTest
@DataJdbcTest
@DataJpaTest
@DataLdapTest
@DataMongoTest
@DataNeo4jTest
@DataR2dbcTest
@DataRedisTest

Test slice
@GraphQlTest
@JdbcTest
@JooqTest
@JsonTest
@RestClientTest
@WebFluxTest
@WebMvcTest
@WebServiceClientTest
@WebServiceServerTest

org.springframework.boot.test.autoconfigure.web.servlet

Annotation Type WebMvcTest

```
@Target(value=TYPE)
 @Retention(value=RUNTIME)
 @Documented
 @Inherited
 @BootstrapWith(value=org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTestContextBootstrapper.class)
 @ExtendWith(value=org.springframework.test.context.junit.jupiter.SpringExtension.class)
 @OverrideAutoConfiguration(enabled=false)
 @TypeExcludeFilters(value=WebMvcTypeExcludeFilter.class)
 @AutoConfigureCache
 @AutoConfigureWebMvc
 @AutoConfigureMockMvc
 @ImportAutoConfiguration
public @interface WebMvcTest
```

Annotation that can be used for a Spring MVC test that focuses **only** on Spring MVC components.

Using this annotation will disable full auto-configuration and instead apply only configuration relevant to MVC tests (i.e. `@Controller`, `@ControllerAdvice`, `@JsonComponent`, `Converter/GenericConverter`, `Filter`, `WebMvcConfigurer` and `HandlerMethodArgumentRes`
but not `@Component`, `@Service` or `@Repository` beans).

By default, tests annotated with `@WebMvcTest` will also auto-configure Spring Security and `MockMvc` (include support for HtmlUnit WebClient and Selenium WebDriver). For more fine-grained control of MockMVC the `@AutoConfigureMockMvc` annotation can be used.

Typically `@WebMvcTest` is used in combination with `@MockBean` or `@Import` to create any collaborators required by your `@Controller` beans.

If you are looking to load your full application configuration and use MockMVC, you should consider `@SpringBootTest` combined with `@AutoConfigureMockMvc` rather than this annotation.

When using JUnit 4, this annotation should be used in combination with `@RunWith(SpringRunner.class)`.

Since:

`1.4.0`

Author:

`Phillip Webb, Artsiom Yudovin`

See Also:

`AutoConfigureWebMvc`, `AutoConfigureMockMvc`, `AutoConfigureCache`

Optional Element Summary

Optional Elements

Modifier and Type	Optional Element and Description
`Class<?>[]`	`controllers`

```java
@WebMvcTest(CoffeeController.class)
class CoffeeControllerTest {

    @Autowired
    private MockMvc mvc;

    @MockBean
    private CoffeeRepository coffeeRepository;

    @Test
    void findAllShouldReturn200Status() throws Exception {
        Coffee coffee = new Coffee(null,"TEST", Size.TALL);
        given(coffeeRepository.findAll()).willReturn(List.of(coffee));
        mvc.perform(get("/api/coffee"))
                .andExpect(MockMvcResultMatchers.status().isOk())
                .andExpect(MockMvcResultMatchers.jsonPath("$.*",hasSize(1)));
    }

}
```

Problem

I want to write tests against the same database we use in production but I need that process to be fast and easy.

Solution

Testcontainers is a Java library that supports JUnit tests, providing lightweight, throwaway instances of common databases, Selenium web browsers, or anything else that can run in a Docker container.

Testing is important!

Testing can be hard!

“You have to make the right thing to do, the easy thing to do.”

- Cora Iberkleid

```java
@SpringBootTest
@Testcontainers
class BookRepositoryIntegrationTest {

    @Container
    static PostgreSQLContainer psql = new PostgreSQLContainer("postgres:alpine");

    @Autowired
    BookRepository repository;

    @Test
    void findAll_shouldReturn1Books() {
        Integer mappedPort = psql.getMappedPort(5432);
        System.out.println("Mapped Port: " + mappedPort);
        assertEquals(1, repository.findAll().size());
    }

}
```

Production


Production

Problem

Your organization uses Docker to containerize your applications.

Solution

You can leverage Cloud Native Buildpacks to simplify and ensure consistency in your applications ecosystem.

2.2. Building a Native Image Using Buildpacks

Spring Boot includes buildpack support for native images directly for both Maven and Gradle. This means you can just type a single command and quickly get a sensible image into your locally running Docker daemon. The resulting image doesn't contain a JVM, instead the native image is compiled statically. This leads to smaller images.

Note

The builder used for the images is `paketobuildpacks/builder:tiny`. It has small footprint and reduced attack surface, but you can also use `paketobuildpacks/builder:base` or `paketobuildpacks/builder:full` to have more tools available in the image if required.

2.2.1. System Requirements

Docker should be installed. See Get Docker for more details. Configure it to allow non-root user if you are on Linux.

Note

You can run `docker run hello-world` (without `sudo`) to check the Docker daemon is reachable as expected. Check the Maven or Gradle Spring Boot plugin documentation for more details.

Tip

On macOS, it is recommended to increase the memory allocated to Docker to at least `8GB`, and potentially add more CPUs as well. See this Stack Overflow answer for more details. On Microsoft Windows, make sure to enable the Docker WSL 2 backend for better performance.

2.2.2. Using Maven

To build a native image container using Maven you should ensure that your `pom.xml` file uses the `spring-boot-starter-parent` and the `org.graalvm.buildtools:native-maven-plugin`. You should have a `<parent>` section that looks like this:

```xml
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.0.5</version>
</parent>
```

You additionally should have this in the `<build>` `<plugins>` section:

Problem

You want your application to startup faster and take up less memory.

Solution

You can compile your application to a native executable using Spring Native which is built in to Spring Boot 3.

Back to index

1. Introducing GraalVM Native Images
   1.1. Key Differences with JVM Deployments
   1.2. Understanding Spring Ahead-of-Time Processing
2. Developing Your First GraalVM Native Application
3. Testing GraalVM Native Images
4. Advanced Native Images Topics
5. What to Read Next

GraalVM Native Image Support

GraalVM Native Images are standalone executables that can be generated by processing compiled Java applications ahead-of-time. Native Images generally have a smaller memory footprint and start faster than their JVM counterparts.

1. Introducing GraalVM Native Images

GraalVM Native Images provide a new way to deploy and run Java applications. Compared to the Java Virtual Machine, native images can run with a smaller memory footprint and with much faster startup times.

They are well suited to applications that are deployed using container images and are especially interesting when combined with "Function as a service" (FaaS) platforms.

Unlike traditional applications written for the JVM, GraalVM Native Image applications require ahead-of-time processing in order to create an executable. This ahead-of-time processing involves statically analyzing your application code from its main entry point.

A GraalVM Native Image is a complete, platform-specific executable. You do not need to ship a Java Virtual Machine in order to run a native image.

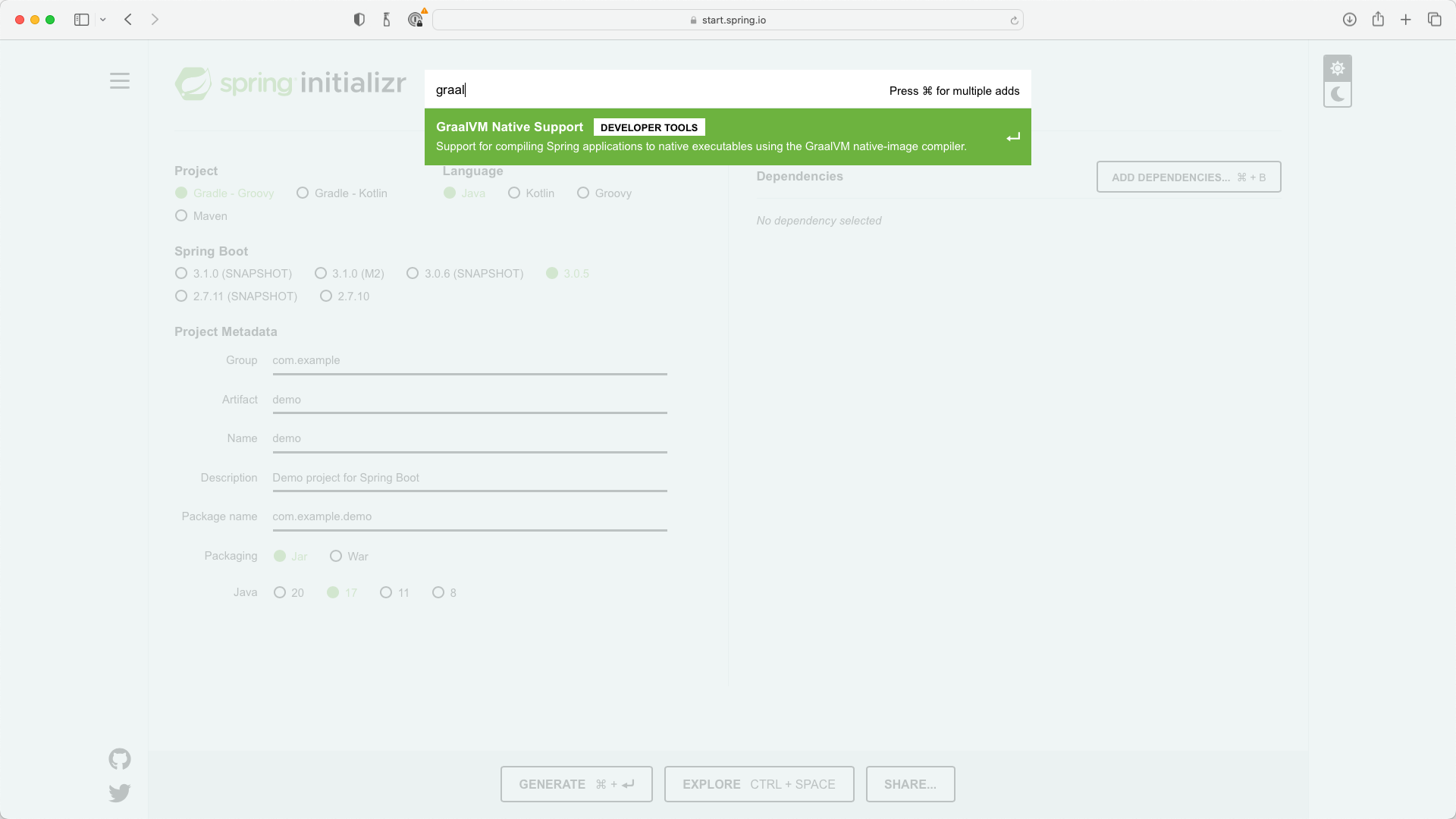
Tip

If you just want to get started and experiment with GraalVM you can skip ahead to the "Developing Your First GraalVM Native Application" section and return to this section later.

1.1. Key Differences with JVM Deployments

The fact that GraalVM Native Images are produced ahead-of-time means that there are some key differences between native and JVM based applications. The main differences are:

- Static analysis of your application is performed at build-time from the `main` entry point.
- Code that cannot be reached when the native image is created will be removed and won't be part of the executable.
- GraalVM is not directly aware of dynamic elements of your code and must be told about reflection, resources, serialization, and dynamic proxies.


start.spring.io
spring initializr
graal
Press ⌘ for multiple adds
GraalVM Native Support
DEVELOPER TOOLS
Support for compiling Spring applications to native executables using the GraalVM native-image compiler.
Project
Gradle - Groovy
Gradle - Kotlin
Maven
Language
Java
Kotlin
Groovy
Spring Boot
3.1.0 (SNAPSHOT)
3.1.0 (M2)
3.0.6 (SNAPSHOT)
3.0.5
2.7.11 (SNAPSHOT)
2.7.10
Project Metadata
Group
com.example
Artifact
demo
Name
demo
Description
Demo project for Spring Boot
Package name
com.example.demo
Packaging
Jar
War
Java
20
17
11
8
Dependencies
ADD DEPENDENCIES... ⌘ + B
No dependency selected
GENERATE ⌘ + ↵
EXPLORE CTRL + SPACE
SHARE...

Spring Projects


Spring Boot


Spring Cloud


Spring Framework

Spring Cloud Data Flow

Spring Tool Suite


Spring LDAP


Spring Cloud Gateway


Spring Security


Spring Data


Spring Batch


Spring Integration


Project Reactor


Spring Kafka


Spring for GraphQL


Spring Web Services


Spring Web Flow


Spring Hateoas


Spring AMQP

Thank you!

Nate (@ntschutta) www.ntschutta.io

Dan (@therealdanvega) www.danvega.dev