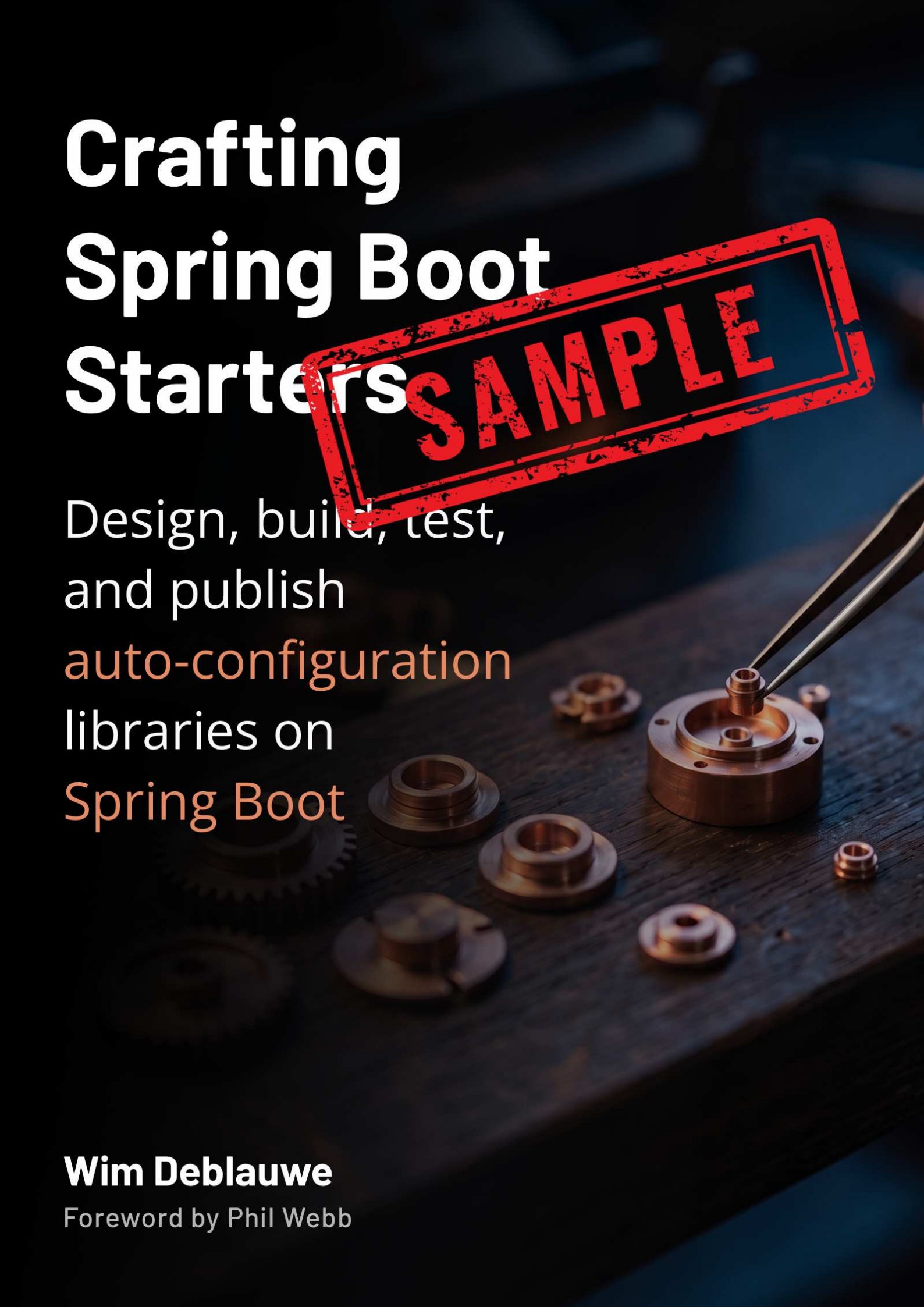
Crafting Spring Boot Starters

Design, build, test, and publish auto-configuration libraries on Spring Boot



Wim Deblauwe

Foreword by Phil Webb

Crafting Spring Boot Starters (Sample)

Build production-grade Spring Boot libraries

Wim Deblauwe

Version 1.0.0-SNAPSHOT, 2026-08-19

Table of Contents

Foreword

A favorite part of working on Spring Boot is seeing how the project is actually used in real-world applications. Not all software is fortunate enough to be so widely adopted and deployed!

For many developers, using Spring Boot directly is just the first step in their journey. They don't want to stop there. They want to extend the project and make it their own. They want to help their teammates by offering new features, or sensible defaults, or simple configuration for their domain.

Going beyond the basics isn't easy! The reference documentation can't go that deep, and trying to read the code without a guide is very hard. That's why I was delighted to hear that Wim was writing a book on this very topic.

Wim has been an active member of the Spring Boot community for many years and has real experience to share from his own open-source projects. If you want an approachable, accurate and insightful guide to extending Spring Boot, this is the book for you!

Thank you Wim for all your contributions, and for investing so much time in this book. I'm confident that your readers will find it incredibly useful and return to it again and again.

~ Phil Webb
current lead of the Spring Boot project

Sample introduction

This book is an excerpt from the Crafting Spring Boot Starters book by Wim Deblauwe.

It includes the first two chapters of the book, which cover the foundations of building Spring Boot starters and the core concepts of auto-configuration.

See the full table of contents and get the book at https://leanpub.com/crafting-spring-boot-starters

Chapter 1. What is a Spring Boot library?

Every Spring Boot application depends on dozens of libraries you did not write. You add `spring-boot-starter-webmvc` and you get an embedded server. You add `spring-boot-starter-data-jpa` and you get a configured `EntityManager`. You add a vendor starter and, after setting a few properties, you are talking to their service.

Each one of those was written by someone who thought about the developer using it.

This book is for that someone.

It is for the developer who needs to share a feature across applications, not build one inside a single application. That changes how you write code. Your audience is no longer you and your team. It is every team that pulls in your library for the next several years.

1.1. The starter mental model

Spring Boot's promise is well known: add a dependency, set a few properties, run the application. From the user's seat that feels like magic. From the author's seat, it is knowing what files to place where, and what annotations to use when.

Spring Boot's tagline is *convention over configuration*. Library authors should strive to honor that phrase. The *convention* part means that there should be as little configuration as possible needed for the common case. The *configuration* part means that users should be able to configure and override almost any part of the library if they need to.

There are two perspectives to keep in mind throughout this book.

This is what your user sees:

- One dependency, often a single line.
- A small block of properties under a namespace, like `mylibrary.kafka.*`.
- Beans they can inject. Beans they can replace.
- A README that explains exactly enough to get started.

This is what you write to make that happen:

- One or more `@AutoConfiguration` classes.
- A handful of conditional annotations on every bean.
- A `@ConfigurationProperties` record with sensible defaults.
- A `spring-configuration-metadata.json` so IDEs autocomplete those properties.
- Tests that prove the configuration behaves correctly under every combination of conditions.
- A sample application that demonstrates the happy path.

1.2. Auto-configuration vs `@Configuration`

The first instinct when sharing Spring beans across projects is often to write a `@Configuration` class, drop it in a shared module, and tell colleagues to `@Import` it. That works, up to a point. But that is not what you need to do for the official Spring Boot starters. So your own starter should also not require this.

A plain `@Configuration` class is written for your application. You know which beans exist, which profiles are active, what the classpath looks like, and what the user wants. The user is you.

An auto-configuration class is written for someone else's application. You do not know whether they are on a servlet stack or a reactive one. You do not know whether the dependency you would like to use is on their classpath. You do not know whether they have already defined a bean of the type you wanted to create. You do not know whether they even want your feature switched on today.

Every one of those unknowns becomes a condition in your code.

Three principles allow you to deal with those unknowns:

1. Activate only when it makes sense. If the user has not put the relevant library on the classpath, your beans should not exist. If they have explicitly disabled your feature, your beans should not exist. Conditions are how you express this.
2. Step aside when the user supplies their own bean. If the user has defined their own `RestClient`, do not overwrite it. `@ConditionalOnMissingBean` is how you express this.

> Stepping aside works because of when your beans are defined. Spring Boot processes the user's own `@Configuration` classes first, then applies auto-configuration last. Your library's beans are defined after the user's, so `@ConditionalOnMissingBean` can see a bean the user already declared and skip creating its own. This is also why `@ConditionalOnMissingBean` and `@ConditionalOnBean` belong on auto-configuration classes and nowhere else, a point Chapter 4 returns to.

3. Fail with actionable errors. When something is genuinely misconfigured, the user should see a sentence that tells them what to do, not a stack trace from deep inside your library.

Most of the patterns in this book are, at heart, a way to make one of those three principles concrete.

1.3. Who is your user?

It helps to picture the person who will adopt your library.

They are a Java developer on a Spring Boot project. They are probably in the middle of debugging something unrelated when they hit the issue your library solves. They scan your README for thirty seconds, looking for a dependency snippet and a property block. If those two things let them make progress, they keep reading. If not, they close the tab.

Once your library is in their build, they forget about it. That is the goal. They should not have to think about your library again until they need to customize something, or something has gone wrong.

In the customization case, they look for a property first. If that does not work, or does not allow them to do what they need to do, then they will look in the documentation for a customizer interface or an overridable bean second. In the something-has-gone-wrong case, they read whatever error message you produce.

1.4. When *not* to write a library

Not every piece of shared code should become a starter. The decision to write a starter brings permanent overhead: versioning, documentation, compatibility windows, releases, support. Before taking that on, check whether something smaller would do.

A useful distinction is between *shared code* and *shared behavior*.

Shared code

A function, a class, or a small abstraction that several applications happen to use. A utility for parsing a particular header. A value object for a domain concept. These belong in a plain JAR. No auto-configuration, no starter, no Bill of Materials (BOM). Add the dependency, import the class, use it.

Shared behavior

A feature that activates itself once the dependency is on the classpath. A Kafka consumer infrastructure that wires producers, error handlers, and metrics. A library that publishes errors to a central system whenever an exception propagates out of a controller. These need auto-configuration, because the value is in the user *not* having to wire anything by hand.

A simple heuristic: if every consumer of the code would write the same `@Configuration` class, the time has come for a starter. If every consumer would use the code differently, leave it as a plain JAR.

The most common mistake is over-applying the starter pattern. A team writes one auto-configured starter, enjoys the experience, and then turns every internal JAR into one. The result is a build full of starters that pull in transitive dependencies nobody asked for and contribute beans nobody uses. Reach for the starter pattern when the behavior is genuinely shared and genuinely worth activating by default.

1.5. Anatomy of a starter

A Spring Boot starter is, at its simplest, a single Maven module. That module contains an auto-configuration class, a `@ConfigurationProperties` record, an `AutoConfiguration.imports` file, and whatever supporting code the feature needs. The consumer adds the JAR to their build, sets a few properties, and the feature works.

For most libraries, one module is enough. That is the default in this book.

You will sometimes see Spring Boot's own starters split across two or three modules. The split exists because Spring Boot ships hundreds of starters that share infrastructure, and because some users want fine-grained control over their classpath. For most libraries, including internal team libraries, focused open-source starters, and the kind of thing you are likely to write, the split is overkill.

The full split looks like this:

`library-autoconfigure`

The brains. Auto-configuration classes, properties, conditional annotations, metadata.

`library-starter`

A thin module with no Java code. It depends on `library-autoconfigure` and on any third-party libraries the feature needs. Consumers depend on this.

`library-dependencies`

A Bill of Materials, packaged as `pom`. It declares versions for every module the library publishes.

The split is worth the effort when:

- You want to ship more than one starter on top of one auto-configuration module. For example, a messaging library with separate `kafka-starter` and `kafka-streams-starter` artifacts, both backed by the same brains.
- You want advanced users to be able to opt into your auto-configuration without taking the full

transitive dependency set.

- You ship a separate test starter alongside the main library.

If none of those apply, stick with a single module.

The next chapter builds a complete single-module starter end-to-end. Chapter 27 returns to the multi-module split in detail.

1.6. A taxonomy of libraries

It helps to know roughly what kind of library you are writing, because the design pressures vary. Four broad categories cover most cases.

Infrastructure adapters

The library wraps a piece of infrastructure: a message broker, an HTTP API, a storage system, a third-party SaaS. The user wants to send and receive messages, or call methods, without thinking about connections, serialization, retries, or shutdown ordering. Examples include Kafka starters, JMS starters, AMQP starters, S3 clients, and SaaS SDKs. The design pressures here are around connection lifecycles, configuration of credentials and endpoints, and graceful failure when the infrastructure is unavailable.

Cross-cutting concerns

The library does not introduce a new feature so much as decorate existing code with a useful property. Observability libraries add tracing and metrics to beans you already have. Security libraries add filters and authentication to your web stack. Error-reporting libraries hook into exception handlers across the application. The design pressures here are around being transparent (the user should not change their code) and composing politely with other cross-cutting libraries.

Domain primitives

The library ships shared types: value objects, validators, common DTOs, identifiers, perhaps a small set of utilities. These often do *not* need auto-configuration, and might be better off as a plain JAR. They become starter-worthy only when activating them requires Spring beans, for example registering a global Jackson module or a `Converter`. The design pressures here are around stability: domain primitives are depended on widely and become very hard to change.

Patterns

The library packages a well-known recipe: a transactional outbox, a retry framework, an idempotency store, feature flags, rate limiting. Each application could implement the pattern itself, but slightly differently each time, with subtle bugs. The design pressures here are around making the pattern's invariants robust regardless of how the user plugs in their own pieces.

Summary

In this chapter, you learned:

- The promise that Spring Boot starters make to their users, and the work that promise hides from view.
- The difference between a plain `@Configuration` class and an auto-configuration class, and the three principles that follow from writing for someone else's application.
- When a starter is the right answer and when a plain JAR is enough.
- The anatomy of a starter.

- Four broad categories of libraries.

The next chapter builds a small starter end-to-end, giving the rest of the book something concrete to refer back to.

Chapter 2. A tour of a minimal starter

The example in this chapter is small on purpose. It has one job, one bean, and two properties, and it fits in a single Maven module. By the end of the chapter you will have a starter installed in your local Maven repository, and a sample application that consumes it. Later chapters revisit each piece in depth.

2.1. What this chapter builds

The example is a `request-id` starter.

Many applications want every incoming HTTP request to carry a correlation identifier. When a request hits the application, the starter checks for an `X-Request-Id` header. If the header is missing, the starter generates a UUID and uses that as the identifier. The identifier is added to the response (so the caller can see it) and placed into the SLF4J Mapped Diagnostic Context so log lines emitted during the request automatically include it.

That is small enough to read in a single sitting, and big enough to need every part of a minimal starter:

- Two properties to let the user change the header name and the MDC key.
- A filter that does the work.
- An auto-configuration class that wires the filter only when it makes sense.
- A POM and an `AutoConfiguration.imports` file so Spring Boot finds it.

2.2. Project layout

The starter is a single Maven module. A sample application lives next to it for demonstration, built separately.

```
/
├── pom.xml                # starter POM
├── src/main/
│   ├── java/com/example/requestid/
│   │   ├── RequestIdProperties.java
│   │   ├── RequestIdFilter.java
│   │   └── RequestIdAutoConfiguration.java
│   └── resources/META-INF/spring/
│       └──
org.springframework.boot.autoconfigure.AutoConfiguration.imports
└── sample/                # consumer application, built standalone
    ├── pom.xml
    └── src/main/...
```

2.3. The POM

The starter is one module, so it gets one POM.

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example.requestid</groupId>
    <artifactId>request-id-spring-boot-starter</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <name>request-id-spring-boot-starter</name>
    <description>Adds an X-Request-Id header and MDC entry to every HTTP
request.</description>

    <properties>
        <project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
        <maven.compiler.release>21</maven.compiler.release>
        <spring-boot.version>4.0.6</spring-boot.version>
    </properties>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-dependencies</artifactId>
                <version>${spring-boot.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-webmvc</artifactId>
        </dependency>
    </dependencies>

    <build>
        <plugins>
```

```
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <annotationProcessorPaths>
                        <path>
                            <groupId>org.springframework.boot</groupId>
                            <artifactId>spring-boot-configuration-
processor</artifactId>
                        </path>
                    </annotationProcessorPaths>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

A few things to notice:

- `spring-boot-dependencies` is imported in `<dependencyManagement>` so the starter inherits Spring Boot's curated versions of every dependency.
- `spring-boot-starter-webmvc` is a hard dependency because the request-id starter is for servlet web applications. A consumer who adds the starter to their build gets the web stack along with it, which is the right behaviour for a library that exists to do something with HTTP requests.
- `spring-boot-configuration-processor` is wired in through the `maven-compiler-plugin`'s `<annotationProcessorPaths>`. It runs at compile time and generates `META-INF/spring-configuration-metadata.json` from the `@ConfigurationProperties` record, which is what makes property names autocomplete in the user's IDE. Since JDK 21, `javac` warns that annotation processors discovered through the classpath may stop being supported, so the processor-path form is the right approach going forward.
- Java 21 is the baseline. That is a middle ground between Spring Boot 4's Java 17 minimum and the current Java 25 LTS, which some teams may not have adopted yet.

2.4. The properties record

The starter has two properties.

src/main/java/com/example/requestid/RequestIdProperties.java

```
package com.example.requestid;

import
org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.bind.DefaultValue;

@ConfigurationProperties("request-id")
public record RequestIdProperties(
```

```
        @DefaultValue("X-Request-Id") String headerName,
        @DefaultValue("requestId") String mdcKey) {
}
```

A few things:

- `@ConfigurationProperties("request-id")` binds anything under `request-id.*` in `application.properties` or `application.yaml` to this record.
- The record uses `@DefaultValue` on each parameter, so the user can omit them entirely.
- The record is immutable. You cannot accidentally mutate properties at runtime, and Spring's binder works directly through the canonical constructor.

Properties as records is the default style throughout this book. Chapter 7 covers records, validation, and nested properties in depth.

2.5. The filter

The filter does the actual work.

src/main/java/com/example/requestid/RequestIdFilter.java

```
package com.example.requestid;

import java.io.IOException;
import java.util.UUID;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

import org.slf4j.MDC;
import org.springframework.web.filter.OncePerRequestFilter;

public class RequestIdFilter extends OncePerRequestFilter {

    private final String headerName;
    private final String mdcKey;

    public RequestIdFilter(String headerName, String mdcKey) {
        this.headerName = headerName;
        this.mdcKey = mdcKey;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request,
```

```
                                    HttpServletResponse response,
                                    FilterChain filterChain) throws
ServletException, IOException {

        String requestId = request.getHeader(headerName);
        if (requestId == null || requestId.isBlank()) {
            requestId = UUID.randomUUID().toString();
        }

        response.setHeader(headerName, requestId);
        MDC.put(mdcKey, requestId);
        try {
            filterChain.doFilter(request, response);
        } finally {
            MDC.remove(mdcKey);
        }
    }
}
```

It extends `OncePerRequestFilter` from Spring, which guarantees the filter runs exactly once per request. The behaviour matches the description above: read the header, generate one if missing, set it on the response, put it on the MDC, clean up afterwards. The cleanup is done in a `finally` block to ensure the MDC is cleared even if the request handling throws an exception.

The filter carries no annotations of its own. The auto-configuration class, covered next, is what decides when to register it as a bean.

2.6. The auto-configuration class

src/main/java/com/example/requestid/RequestIdAutoConfiguration.java

```
package com.example.requestid;

import jakarta.servlet.Filter;

import org.springframework.boot.autoconfigure.AutoConfiguration;
import
org.springframework.boot.autoconfigure.condition.ConditionalOnClass;
import
org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBea
n;
import
org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;
import
org.springframework.boot.autoconfigure.condition.ConditionalOnWebApplica
tion;
```

```
import
org.springframework.boot.context.properties.EnableConfigurationPropertie
s;
import org.springframework.context.annotation.Bean;

@AutoConfiguration
@ConditionalOnClass(Filter.class)
@ConditionalOnWebApplication(type = ConditionalOnWebApplication.Type
.SERVLET)
@ConditionalOnProperty(name = "request-id.enabled", matchIfMissing =
true)
@EnableConfigurationProperties(RequestIdProperties.class)
public class RequestIdAutoConfiguration {

    @Bean
    @ConditionalOnMissingBean
    public RequestIdFilter requestIdFilter(RequestIdProperties
properties) {
        return new RequestIdFilter(properties.headerName(), properties
.mdcKey());
    }
}
```

There is more going on here, so each annotation deserves a closer look.

`@AutoConfiguration`

Marks this class as an auto-configuration. In Spring Boot, `@AutoConfiguration` replaces `@Configuration` for library code. It carries `@Configuration(proxyBeanMethods = false)` semantics and signals that the class is intended for discovery, not for `@Import`.

`@ConditionalOnClass(Filter.class)`

The auto-configuration only activates when `jakarta.servlet.Filter` is on the classpath. If the consumer is not a servlet application, none of this code runs. Because the starter declares a hard dependency on `spring-boot-starter-webmvc`, `Filter` is almost always present, so this guard looks redundant here. Keep it anyway: it states plainly that the configuration is built on the servlet `Filter` API, and it earns its place once the starter is split into a separate autoconfigure module (Chapter 27), where the web dependency becomes optional and `Filter` is no longer guaranteed.

`@ConditionalOnWebApplication(type = SERVLET)`

This annotation tightens the previous condition: the auto-configuration applies only to servlet web applications, not to reactive applications or non-web contexts.

`@ConditionalOnProperty(name = "request-id.enabled", matchIfMissing = true)`

The user can switch the feature off by setting `request-id.enabled=false`. The `matchIfMissing = true` means the feature is on by default. With an empty `havingValue`, the condition matches any value that is not `false`. Chapter 5 covers `@ConditionalOnBooleanProperty`, a more direct way to express this on/off flag, along with the full set of conditional annotations.

`request-id.enabled` is not part of the properties record, so the annotation processor never sees it and the IDE will not autocomplete it. Properties referenced only in `@ConditionalOnProperty` have to be declared by hand, which Chapter 10 covers.

`@EnableConfigurationProperties(RequestIdProperties.class)`
: Registers the properties record as a bean and binds its values from the environment.

Inside the class, the single `@Bean` method creates the filter. The `@ConditionalOnMissingBean` annotation is the user-override hook: if the consumer has declared their own `RequestIdFilter`, the starter's filter is not registered. This is the *step aside* principle from Chapter 1, written down as a single annotation.

2.7. Telling Spring Boot about the auto-configuration

Annotating a class with `@AutoConfiguration` is not enough to be picked up automatically. Spring Boot discovers auto-configurations by reading a single text file inside the JAR at `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports`

src/main/resources/META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports

```
com.example.requestid.RequestIdAutoConfiguration
```

One fully qualified class name per line. That is the whole file.

The dots in `org.springframework.boot.autoconfigure.AutoConfiguration.imports` are part of the file name, not directory separators. The file lives directly inside `META-INF/spring/`.

2.8. Trying it out

The repository ships with a sample application under `sample/`. It is a plain Spring Boot project with a single controller and a dependency on the starter.

The sample's POM uses `spring-boot-starter-parent` as its build parent and depends on the starter by name and version:

sample/pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>org.springframework.boot</groupId>
```

```
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>4.0.6</version>
        <relativePath/>
    </parent>

    <groupId>com.example.requestid.sample</groupId>
    <artifactId>request-id-sample</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <name>request-id-sample</name>
    <description>Sample application that consumes the request-id
starter</description>

    <properties>
        <java.version>21</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>com.example.requestid</groupId>
            <artifactId>request-id-spring-boot-starter</artifactId>
            <version>0.0.1-SNAPSHOT</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

The application has two source files. A `@SpringBootApplication` class:

sample/src/main/java/com/example/requestid/sample/SampleApplication.java

```
package com.example.requestid.sample;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
```

```
public class SampleApplication {

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

And a controller:

sample/src/main/java/com/example/requestid/sample/HelloController.java

```
package com.example.requestid.sample;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {

    private static final Logger logger = LoggerFactory.getLogger
(HelloController.class);

    @GetMapping("/hello")
    public String hello() {
        logger.info("handling /hello");
        return "hello";
    }
}
```

The `application.properties` file configures the log pattern so that the request identifier from the MDC appears next to the logging level (DEBUG, INFO, ...) in every log line:

sample/src/main/resources/application.properties

```
logging.pattern.level=%5p [%X{requestId:-}]

# Uncomment to override the defaults shipped with the starter.
# request-id.header-name=X-Correlation-Id
# request-id.mdc-key=correlationId
# request-id.enabled=false
```

The sample properties file also shows how you can override the configuration defaults.

`mvn install` builds the starter JAR and copies it into your local Maven repository (typically `~/.m2/repository`). That is how the sample application finds it: the sample declares a dependency on `com.example.requestid:request-id-spring-boot-starter:0.0.1-SNAPSHOT`, and Maven resolves that artifact from the local repository. For a real library you would publish to a remote repository instead; Chapter 29 covers that.

To run the example, install the starter into your local Maven repository first, then run the sample:

```
mvn install
mvn -f sample/pom.xml spring-boot:run
```

In another terminal, send a request:

```
curl -i http://localhost:8080/hello
```

The response includes a freshly generated `X-Request-Id` header:

```
HTTP/1.1 200
X-Request-Id: 4f9c2e8a-1d3b-4a7c-9f0e-2b8e4d6a1f3b
Content-Type: text/plain;charset=UTF-8

hello
```

In the application log, the request identifier appears between square brackets, so any log line emitted while the request is being handled carries the correlation ID:

```
 INFO [4f9c2e8a-1d3b-4a7c-9f0e-2b8e4d6a1f3b]
c.e.r.sample.HelloController : handling /hello
```

If you supply your own `X-Request-Id`, the starter uses that one instead:

```
curl -i -H "X-Request-Id: my-custom-id" http://localhost:8080/hello
```

The response echoes `my-custom-id` back, and the same value appears in the log:

```
 INFO [my-custom-id] c.e.r.sample.HelloController : handling /hello
```

Setting `request-id.enabled=false` turns the feature off. Stop the sample and restart it with the property set:

```
mvn -f sample/pom.xml spring-boot:run \
```

```
  -Dspring-boot.run.arguments=--request-id.enabled=false
```

The auto-configuration backs off, so no filter is registered. The response now carries no `X-Request-Id` header, and the MDC placeholder in the log stays empty:

```
HTTP/1.1 200
Content-Type: text/plain;charset=UTF-8

hello
```

2.9. How Spring Boot finds your starter

Here is what happens under the hood when the sample starts.

When the sample application starts, Spring Boot's auto-configuration mechanism reads every `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports` file on the classpath. The starter JAR contributes one such file with one line in it: `com.example.requestid.RequestIdAutoConfiguration`.

Spring Boot loads that class as an auto-configuration source and evaluates its conditions. Because the sample is a servlet web application and `request-id.enabled` is not set to `false`, it creates the `RequestIdFilter` bean. Because the sample does not declare its own filter, `@ConditionalOnMissingBean` does not block the starter's bean. The filter is registered with Spring's filter chain like any other servlet `Filter` bean, and from that point on every request flows through it.

The consumer wrote nothing. That is the whole point of an auto-configuring starter, and the rest of this book is about doing this well across a much wider range of features.

Summary

In this chapter, you learned:

- The shape of a minimal starter: one Maven module containing the auto-configuration, properties, conditional annotations, and the `AutoConfiguration.imports` file that tells Spring Boot to load it.
- How to express configuration with a `@ConfigurationProperties` record and `@DefaultValue`.
- The four most common conditional annotations: `@ConditionalOnClass`, `@ConditionalOnWebApplication`, `@ConditionalOnProperty`, and `@ConditionalOnMissingBean`.
- How `@AutoConfiguration` and the `AutoConfiguration.imports` file work together to register an auto-configuration class.

The next chapter steps back from the code and turns the design principles introduced in Chapter 1 into a checklist you can apply to your own libraries.

Closing

I hope you enjoyed this free sample of the Crafting Spring Boot Starters book.

Keep your learning going by getting the full book at https://leanpub.com/crafting-spring-boot-starters

Happy coding!