
Spring Boot

(/) Java
SPRING BOOT (/CATEGORIES/SPRING-BOOT)

Software Craft
Using Kafka with Spring Boot
Book Reviews

Meta
In this article, we'll look at how to integrate a Spring Boot application with Apache Kafka and s
consuming messages from our application. We'll be going through each section with code exar

(https://github.com/thombergs/code-examples/tree/master/spr boot/spring-boot-kafka) Code Example

This article is accompanied by a working code example on GitHub (https://github.com/thombe examples/tree/master/spring-boot/spring-boot-kafka).

Why Kafka?

Traditional messaging queues like ActiveMQ, RabbitMQ can handle high throughput usually us or background jobs and communicating between services.

Kafka is a stream-processing platform built by LinkedIn and currently developed under the um Software Foundation. Kafka aims to provide low-latency ingestion of large amounts of event da

We can use Kafka when we have to move a large amount of data and process it in real-time. Ar when we want to process user behavior on our website to generate product suggestions or mon by our micro-services.

Kafka is built from ground up with horizontal scaling in mind. We can scale by adding more br Kafka cluster.

Kafka Vocabulary

Let's look at the key terminologies of Kafka:

- **Producer**: A producer is a client that sends messages to the Kafka server to the specified top
- **Consumer**: Consumers are the recipients who receive messages from the Kafka server.
- **Broker**: Brokers can create a Kafka cluster by sharing information using Zookeeper. A broke from producers and consumers fetch messages from the broker by topic, partition, and offs
- **Cluster**: Kafka is a distributed system. A Kafka cluster contains multiple brokers sharing th

- **Topic**: A topic is a category name to which messages are published and from which consun messages.
- **Partition**: Messages published to a topic are spread across a Kafka cluster into several parti can be associated with a broker to allow consumers to read from a topic in parallel.
- **Offset**: Offset is a pointer to the last message that Kafka has already sent to a consumer.

Configuring a Kafka Client

We should have a Kafka server running on our machine. If you don't have Kafka setup on your the Kafka quickstart guide (https://kafka.apache.org/quickstart). Once we have a Kafka server u Kafka client can be easily configured with Spring configuration in Java or even quicker with Sp

Let's start by adding `spring-kafka` dependency to our `pom.xml`:

```
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
  <version>2.5.2.RELEASE</version>
</dependency>
```

Using Java Configuration

Let's now see how to configure a Kafka client using Spring's Java Configuration. To split up resp separated `KafkaProducerConfig` and `KafkaConsumerConfig`.

Let's have a look at the producer configuration first:

```
@Configuration
public class KafkaProducerConfig {

  @Value("${io.reflectoring.kafka.bootstrap-servers}")
  private String bootstrapServers;

  @Bean
  public Map<String, Object> producerConfigs() {
    Map<String, Object> props = new HashMap<>();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
      bootstrapServers);
    props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
      StringSerializer.class);
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
      StringSerializer.class);
    return props;
  }

  @Bean
  public ProducerFactory<String, String> producerFactory() {
    return new DefaultKafkaProducerFactory<>(producerConfigs());
  }

  @Bean
  public KafkaTemplate<String, String> kafkaTemplate() {
    return new KafkaTemplate<>(producerFactory());
  }
}
```

The above example shows how to configure the Kafka producer to send messages. `ProducerFa` for creating Kafka Producer instances.

`KafkaTemplate` helps us to send messages to their respective topic. We'll see more about `Kafka` sending messages section.

In `producerConfigs()` we are configuring a couple of properties:

- `BOOTSTRAP_SERVERS_CONFIG` - Host and port on which Kafka is running.
- `KEY_SERIALIZER_CLASS_CONFIG` - Serializer class to be used for the key.
- `VALUE_SERIALIZER_CLASS_CONFIG` - Serializer class to be used for the value. We are using `S` both keys and values.

Now that our producer config is ready, let's create a configuration for the consumer:


```
@Configuration
public class KafkaConsumerConfig {

  @Value("${io.reflectoring.kafka.bootstrap-servers}")
  private String bootstrapServers;

  @Bean
  public Map<String, Object> consumerConfigs() {
    Map<String, Object> props = new HashMap<>();
    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
      bootstrapServers);
    props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
      StringDeserializer.class);
    return props;
  }

  @Bean
  public ConsumerFactory<String, String> consumerFactory() {
    return new DefaultKafkaConsumerFactory<>(consumerConfigs());
  }

  @Bean
  public KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<String, String>> kafkaListe
    ConcurrentKafkaListenerContainerFactory<String, String> factory =
      new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(consumerFactory());
    return factory;
  }
}
```

We use `ConcurrentKafkaListenerContainerFactory` to create containers for methods annotat `@KafkaListener`. The `KafkaListenerContainer` receives all the messages from all topics or pa thread. We'll see more about message listener containers in the consuming messages section.

Using Spring Boot Auto Configuration

Spring Boot does most of the configuration automatically, so we can focus on building the liste the messages. It also provides the option to override the default configuration through `applica` Kafka configuration is controlled by the configuration properties with the prefix `spring.kafka`

```
spring.kafka.bootstrap-servers=localhost:9092
spring.kafka.consumer.group-id=myGroup
```

Creating Kafka Topics

A topic must exist to start sending messages to it. Let`s now have a look at how we can create K

```
@Configuration
public class KafkaTopicConfig {

  @Bean
  public NewTopic topic1() {
    return TopicBuilder.name("reflectoring-1").build();
  }
  @Bean
  public NewTopic topic2() {
    return TopicBuilder.name("reflectoring-2").build();
  }
  ...
}
```

A `KafkaAdmin` bean is responsible for creating new topics in our broker. **With Spring Boot, a `Ka` automatically registered.**

For a non Spring Boot application we have to manually register `KafkaAdmin` bean:

```
@Bean
KafkaAdmin admin() {
  Map<String, Object> configs = new HashMap<>();
  configs.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, ...);
  return new KafkaAdmin(configs);
}
```

To create a topic, we register a `NewTopic` bean for each topic to the application context. If the to bean will be ignored. We can make use of `TopicBuilder` to create these beans. `KafkaAdmin` als number of partitions if it finds that an existing topic has fewer partitions than `NewTopic.numPa`

Sending Messages

Using `KafkaTemplate`

`KafkaTemplate` provides convenient methods to send messages to topics:

```
@Component
class KafkaSenderExample {

  private KafkaTemplate<String, String> kafkaTemplate;
  ...

  @Autowired
  KafkaSenderExample(KafkaTemplate<String, String> kafkaTemplate, ...) {
    this.kafkaTemplate = kafkaTemplate;
    ...
  }

  void sendMessage(String message, String topicName) {
    kafkaTemplate.send(topicName, message);
  }
  ...
}
```

All we need to do is to call the `sendMessage()` method with the message and the topic name as

Spring Kafka also allows us to configure an async callback:

Java

```
@Component
class KafkaSenderExample {
  ...
  void sendMessageWithCallback(String message) {
    ListenableFuture<SendResult<String, String>> future =
      kafkaTemplate.send(topic1, message);

    future.addCallback(new ListenableFutureCallback<SendResult<String, String>>() {
      @Override
      public void onSuccess(SendResult<String, String> result) {
        LOG.info("Message [{}] delivered with offset {}",
          message,
          result.getRecordMetadata().offset());
      }

      @Override
      public void onFailure(Throwable ex) {
        LOG.warn("Unable to deliver message [{}]. {}",
          message,
          ex.getMessage());
      }
    });
  }
}
```

The `send()` method of `KafkaTemplate` returns a `ListenableFuture<SendResult>`. We can regi `ListenableFutureCallback` with the listener to receive the result of the send and do some wor context.

If we don't want to work with `Future`s, we can register a `ProducerListener` instead:

```
@Configuration
class KafkaProducerConfig {
  @Bean
  KafkaTemplate<String, String> kafkaTemplate() {
  KafkaTemplate<String, String> kafkaTemplate =
    new KafkaTemplate<>(producerFactory());
  ...
  kafkaTemplate.setProducerListener(new ProducerListener<String, String>() {
    @Override
    public void onSuccess(
      ProducerRecord<String, String> producerRecord,
      RecordMetadata recordMetadata) {

      LOG.info("ACK from ProducerListener message: {} offset:  {}",
        producerRecord.value(),
        recordMetadata.offset());
    }
  });
  return kafkaTemplate;
  }
}
```

We configured `KafkaTemplate` with a `ProducerListener` which allows us to implement the `or onError()` methods.

Spring Boot

Java

Software Craft

Book Reviews

Meta

Using RoutingKafkaTemplate

(/)

We can use `RoutingKafkaTemplate` when we have **multiple producers with different configura** select producer at runtime based on the topic name.

```
@Configuration
class KafkaProducerConfig {
  ...

  @Bean
  public RoutingKafkaTemplate routingTemplate(GenericApplicationContext context) {
    // ProducerFactory with Bytes serializer
    Map<String, Object> props = new HashMap<>();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
      bootstrapServers);
    props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
      StringSerializer.class);
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
      ByteArraySerializer.class);
    DefaultKafkaProducerFactory<Object, Object> bytesPF =
      new DefaultKafkaProducerFactory<>(props);
    context.registerBean(DefaultKafkaProducerFactory.class, "bytesPF", bytesPF);

    // ProducerFactory with String serializer
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
      StringSerializer.class);
    DefaultKafkaProducerFactory<Object, Object> stringPF =
      new DefaultKafkaProducerFactory<>(props);

    Map<Pattern, ProducerFactory<Object, Object>> map = new LinkedHashMap<>();
    map.put(Pattern.compile(".*-bytes"), bytesPF);
    map.put(Pattern.compile("reflectoring-.*"), stringPF);
    return new RoutingKafkaTemplate(map);
  }
  ...
}
```

`RoutingKafkaTemplate` takes a map of `java.util.regex.Pattern` and `ProducerFactory` insta messages to the first `ProducerFactory` matching a given topic name. If we have two patterns `reflectoring-.*`, the pattern `reflectoring-.*` should be at the beginning because the `ref.*` "override" it, otherwise.

In the above example, we have created two patterns `.*-bytes` and `reflectoring-.*`. The topi '`-bytes`' and starting with `reflectoring-.*` will use `ByteArraySerializer` and `StringSeria` when we use `RoutingKafkaTemplate` instance.

Consuming Messages

Message Listener

A `KafkaMessageListenerContainer` receives all messages from all topics on a single thread.

A `ConcurrentMessageListenerContainer` assigns these messages to multiple `KafkaMessageLi` instances to provide multi-threaded capability.

Java

Using `@KafkaListener` at Method Level

The `@KafkaListener` annotation allows us to create listeners:

```
@Component
class KafkaListenersExample {

  Logger LOG = LoggerFactory.getLogger(KafkaListenersExample.class);

  @KafkaListener(topics = "reflectoring-1")
  void listener(String data) {
    LOG.info(data);
  }

  @KafkaListener(
    topics = "reflectoring-1, reflectoring-2",
    groupId = "reflectoring-group-2")
  void commonListenerForMultipleTopics(String message) {
    LOG.info("MultipleTopicListener - {}", message);
  }
}
```

To use this annotation we should add the `@EnableKafka` annotation on one of our `@Configurat` requires a listener container factory, which we have configured in `KafkaConsumerConfig.java`

Using `@KafkaListener` will make this bean method a listener and wrap the bean in `MessagingMessageListenerAdapter`. We can also specify multiple topics for a single listener us attribute as shown above.

Using `@KafkaListener` at Class Level

We can also use the `@KafkaListener` annotation at class level. If we do so, we need to specify `@` method level:

```
@Component
@KafkaListener(id = "class-level", topics = "reflectoring-3")
class KafkaClassListener {
  ...

  @KafkaHandler
  void listen(String message) {
    LOG.info("KafkaHandler[String] {}", message);
  }

  @KafkaHandler(isDefault = true)
  void listenDefault(Object object) {
    LOG.info("KafkaHandler[Default] {}", object);
  }
}
```

Spring Boot

(/) Java

Software Craft

Book Reviews

Meta

When the listener receives messages, it converts them into the target types and tries to match t
method signatures to find out which method to call.

In the example, messages of type `String` will be received by `listen()` and type `Object` will b
`listenDefault()`. Whenever there is no match, the default handler (defined by `isDefault=tru`

Consuming Messages from a Specific Partition with an Initial Offset

We can configure listeners to listen to multiple topics, partitions, and a specific initial offset.

For example, if we want to receive all the messages sent to a topic from the time of its creation we can set the initial offset to zero:

```
@Component
class KafkaListenersExample {
  ...

  @KafkaListener(
    groupId = "reflectoring-group-3",
    topicPartitions = @TopicPartition(
      topic = "reflectoring-1",
      partitionOffsets = { @PartitionOffset(
        partition = "0",
        initialOffset = "0") }))
  void listenToPartitionWithOffset(
    @Payload String message,
    @Header(KafkaHeaders.RECEIVED_PARTITION_ID) int partition,
    @Header(KafkaHeaders.OFFSET) int offset) {
      LOG.info("Received message [{}] from partition-{} with offset-{}",
        message,
        partition,
        offset);
  }
}
```

Since we have specified `initialOffset = "0"`, **we will receive all the messages starting from
restart the application.**

We can also retrieve some useful metadata about the consumed message using the `@Header()`

Filtering Messages

Spring provides a strategy to filter messages before they reach our listeners:

```
class KafkaConsumerConfig {
  @Bean
  KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<String, String>>
  kafkaListenerContainerFactory() {
    ConcurrentKafkaListenerContainerFactory<String, String> factory =
      new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(consumerFactory());
    factory.setRecordFilterStrategy(record ->
      record.value().contains("ignored"));
    return factory;
  }
}
```

Spring wraps the listener with a `FilteringMessageListenerAdapter`. It takes an implementati `RecordFilterStrategy` in which we implement the filter method. **Messages that match the filt before reaching the listener.**

In the above example, we have added a filter to discard the messages which contain the word "i

Replying with `@SendTo`

Spring allows sending method's return value to the specified destination with `@SendTo`:

```
@Component
class KafkaListenersExample {
  ...

  @KafkaListener(topics = "reflectoring-others")
  @SendTo("reflectoring-1")
  String listenAndReply(String message) {
    LOG.info("ListenAndReply [{}]", message);
    return "This is a reply sent after receiving message";
  }
}
```

The Spring Boot default configuration gives us a reply template. Since we are overriding the fac above, the listener container factory must be provided with a `KafkaTemplate` by using `setRepl` is then used to send the reply.

In the above example, we are sending the reply message to the topic "reflectoring-1".

Custom Messages

Let's now look at how to send/receive a Java object. We'll be sending and receiving `User` object

```
class User {
  private String name;
  ...
}
```

Configuring JSON Serializer & Deserializer

To achieve this, we must configure our producer and consumer to use a JSON serializer and des

```
@Configuration
class KafkaProducerConfig {
  ...
  @Bean
  public ProducerFactory<String, User> userProducerFactory() {
    ...
    configProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
      JsonSerializer.class);
    return new DefaultKafkaProducerFactory<>(configProps);
  }

  @Bean
  public KafkaTemplate<String, User> userKafkaTemplate() {
    return new KafkaTemplate<>(userProducerFactory());
  }
}
```

```
@Configuration
class KafkaConsumerConfig {
  ...
  public ConsumerFactory<String, User> userConsumerFactory() {
    Map<String, Object> props = new HashMap<>();
    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
    props.put(ConsumerConfig.GROUP_ID_CONFIG, "reflectoring-user");

    return new DefaultKafkaConsumerFactory<>(
      props,
      new StringDeserializer(),
      new JsonDeserializer<>(User.class));
  }

  @Bean
  public ConcurrentKafkaListenerContainerFactory<String, User> userKafkaListenerContainerFactory() {
    ConcurrentKafkaListenerContainerFactory<String, User> factory =
      new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(userConsumerFactory());
    return factory;
  }
  ...
}
```

Spring Kafka provides `JsonSerializer` and `JsonDeserializer` implementations that are base JSON object mapper. It allows us to convert any Java object to `bytes[]`.

In the above example, we are creating one more `ConcurrentKafkaListenerContainerFactory` f In this, we have configured `JsonSerializer.class` as our value serializer in the producer confi `JsonDeserializer<>(User.class)` as our value deserializer in the consumer config.

For this, we are creating a separate Kafka listener container `userKafkaListenerContainerFact` multiple Java object types to be serialized/deserialized, we have to create a listener container fo above.

Sending Java Objects

Now that we have configured our serializer and deserializer, we can send a `User` object using th

```
@Component
class KafkaSenderExample {
  ...
  @Autowired
  private KafkaTemplate<String, User> userKafkaTemplate;

  void sendCustomMessage(User user, String topicName) {
    userKafkaTemplate.send(topicName, user);
  }
  ...
}
```

Receiving Java Objects

We can receive Java objects by using the `@KafkaListener` annotation:

```
@Component
class KafkaListenerExample {

  @KafkaListener(
      topics = "reflectoring-user",
      groupId="reflectoring-user",
      containerFactory="userKafkaListenerContainerFactory")
  void listener(User user) {
    LOG.info("CustomUserListener [{}]", user);
  }
}
```

Since we have multiple listener containers, we are specifying which container factory to use.

If we don't specify the `containerFactory` attribute it defaults to `kafkaListenerContainerFacto` `StringSerializer` and `StringDeserializer` in our case.

Conclusion

In this article, we covered how we can leverage the Spring support for Kafka. Build Kafka based code examples that can help to get started quickly.

You can play around with the code on GitHub (https://github.com/thombergs/code-examples/tr boot/spring-boot-kafka).

Nandan BN

Full stack developer, passionate about technology, working for a better future with the help of digi

(/) Java Software Craft Book Reviews Meta

Get My Book for just $5!

Liked this article? Subscribe to my mailing list to get notified about new content and get my eBook "Get Your Hands Dirty on Clean Architecture" (/e-book/) for just $5!

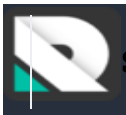
SUBSCRIBE

(/get-your-hands-dirty-o

GET IT AT AMAZON (HTTPS://WWW.AMAZON.COM/GP/PRODUCT/1839211962/REF=AS_LI_TL?IE=UTF8&CAMP=1789&CREATIVE=9325&CREATIVEASIN=1839211962&LINKCODE=AS2&TAG=REFLECTORIN0C-20&LINKID=559E54B6599C4213252259DF28D1D3E3)

Grab My Book for Just $5!


Spring Boot
(/)
Java
Software Craft
Book Reviews
Meta
Clean
A Hands-on
Creating Cle
with Code E



Spring Boot

(/) Java

Software Craft

Book Reviews

Meta

★★★★★ Rated 4.8 stars on Amazon (https://www.amazon.com/gp/product/1839211962/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&cr

Subscribe to my Mailing List and get my book **Get Your Hands Dirty on Clean Architecture (/book/)** for just $5!

GET IT AT AMAZON (HTTPS://W

On This Page

Spring Boot

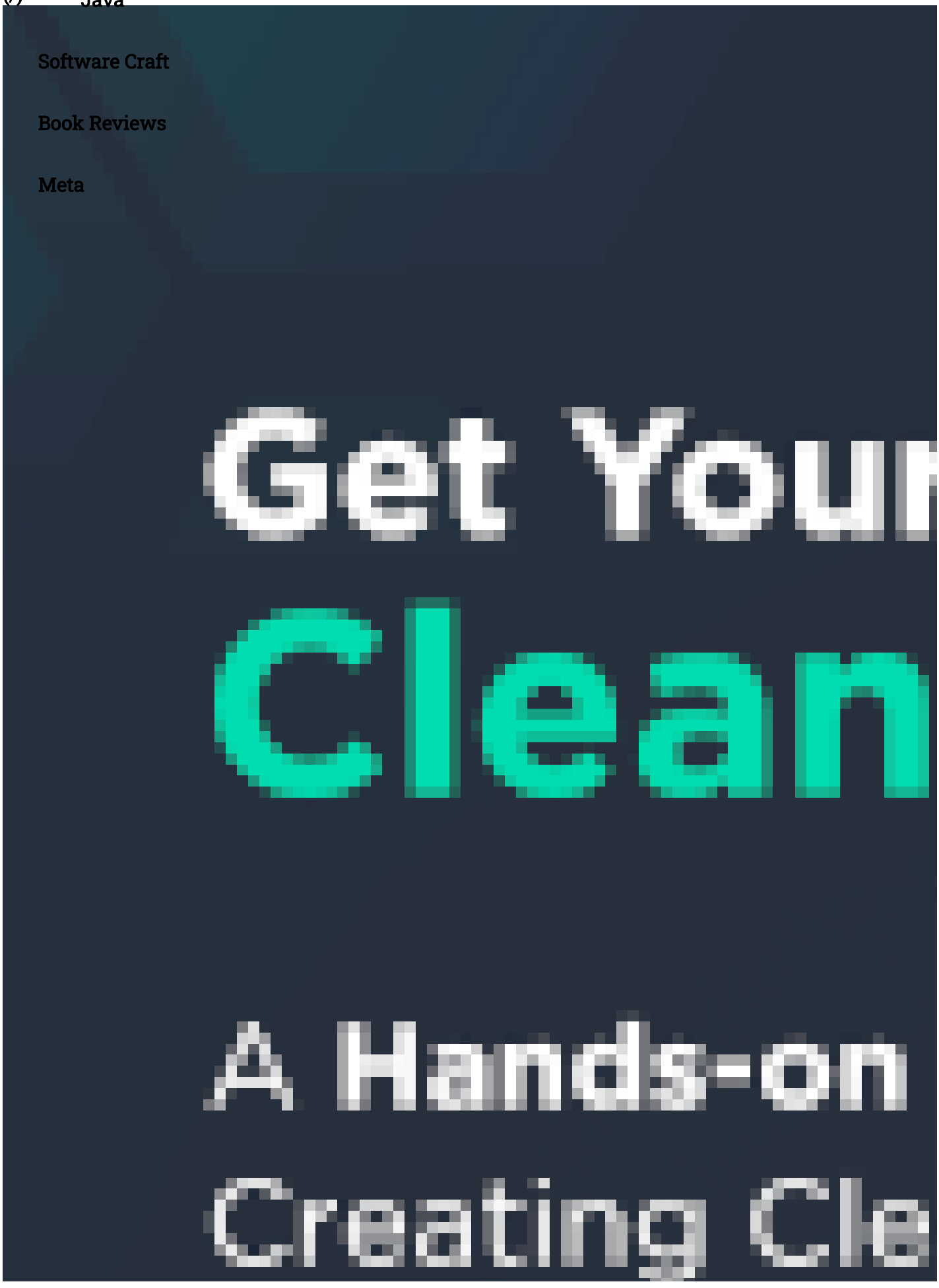
Grab My Book for Just $5!

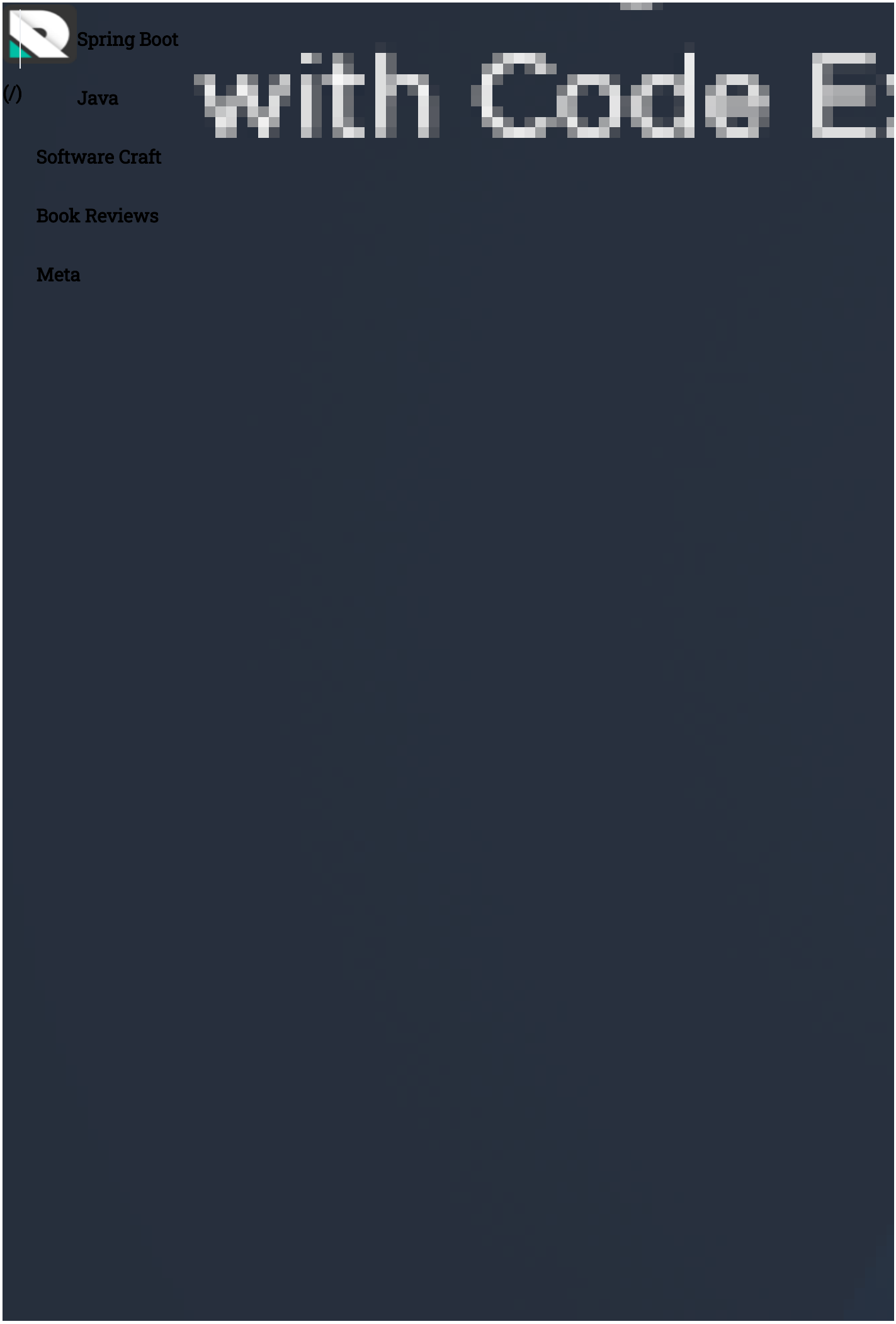
(/) Java

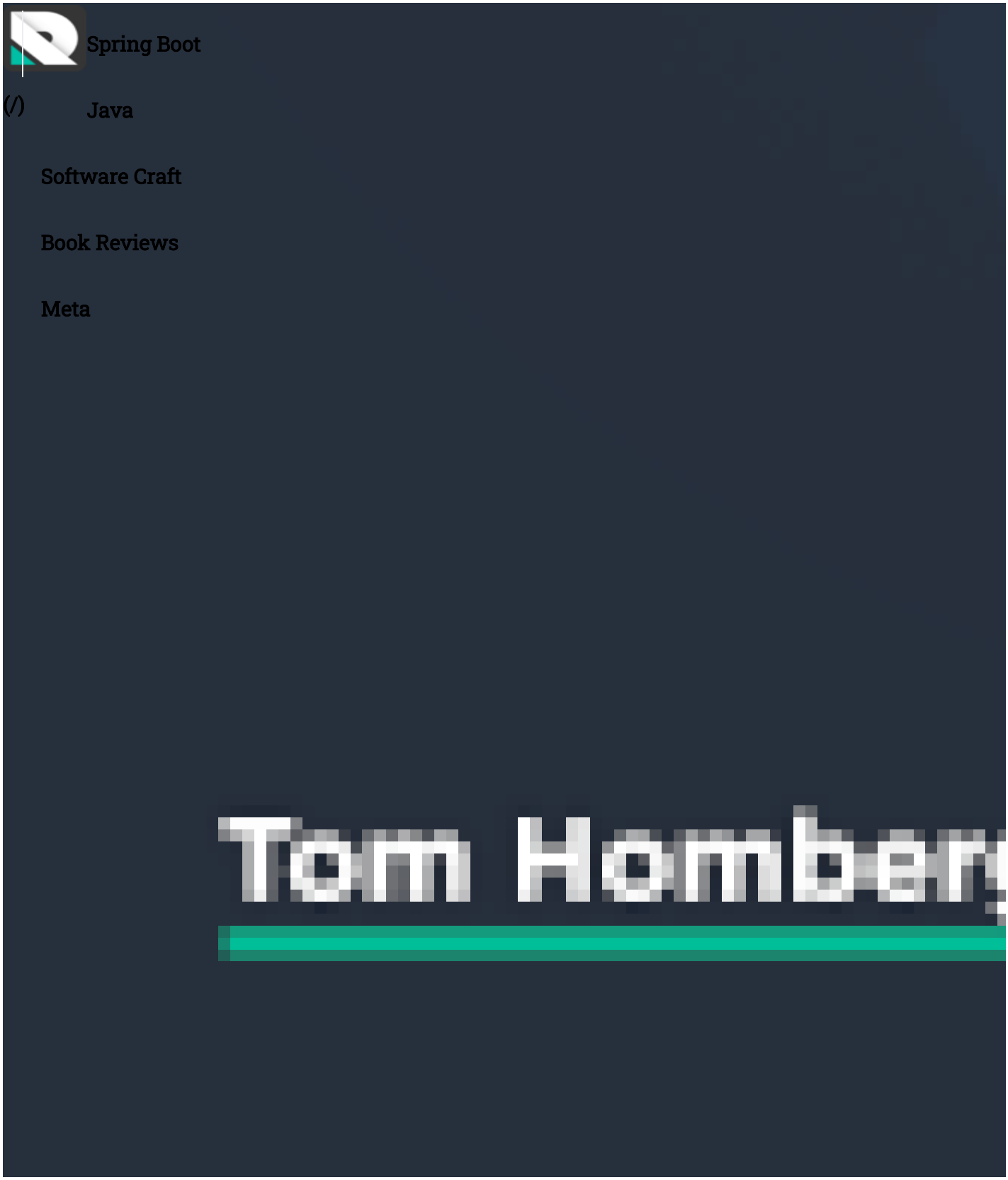
Software Craft

Book Reviews

Meta




Spring Boot
(/)
Java
with Code E
Software Craft
Book Reviews
Meta



★★★★★ Rated 4.8 stars on Amazon (https://www.amazon.com/gp/product/1839211962/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&cr

Subscribe to my Mailing List and get my book **Get Your Hands Dirty on Clean Architecture (/book/)** for just $5!

GET IT AT AMAZON (HTTPS://W

Follow Reflectoring

in
(https://www.linkedin.com/in/tom

Spring Boot

Java

Software Craft

Book Reviews

Meta

Reflectoring

About (/about/)

Atom Feed (/feed.xml)

Meta (/categories/meta)

Privacy Policy (/privacy/)

Write With Me (/write-with-me)

Resources

Book (/get-your-hands-dirty-on-clean-architecture/)

Categories

Spring Boot (/categories/spring-boot/)

Java (/categories/java/)

Software Craft (/categories/craft/)

Book Reviews (/categories/book-reviews)

Programming (/categories/programming)