

# Getting Started With Quarkus

**ALEX SOTO**

DIRECTOR OF DEVELOPER EXPERIENCE, RED HAT

**CONTENTS**

- Key Benefits
- Getting Started
- Key Components
  - JAX-RS
  - Health Checks
  - Security and JWT
  - Docker and Native
  - Container Images
  - And more!
- Conclusion

Quarkus is a Kubernetes-Native Java stack tailored to GraalVM and OpenJDK HotSpot, helping Java programs run 10X faster, while being 100X smaller. Improving the developer experience, Quarkus provides additional features like live reloading and debugging as well as persistence with Panache.

Its integration with the Eclipse MicroProfile specification also makes it the perfect choice for developing microservices and deploying them in Kubernetes.

## KEY BENEFITS

Quarkus offers near-instant scale-up and high-density utilization in container orchestration platforms such as Kubernetes. Many more application instances can be run using the same hardware resources. In Quarkus, classes used only at application startup are invoked at build time and not loaded into the runtime JVM.

Quarkus also avoids reflection as much as possible. These design principles reduce the size and memory footprint of an application running on the JVM. Quarkus' design accounts for native compilation from the onset; optimization for using GraalVM, specifically its native image capability, to compile JVM bytecode to a native machine binary.

Additionally, Quarkus rests on a vast ecosystem of technologies, standards, libraries, and APIs. Developers don't have to spend lots of time learning an entirely new set of APIs and technologies to take advantage of the benefits Quarkus brings to the JVM or native images.

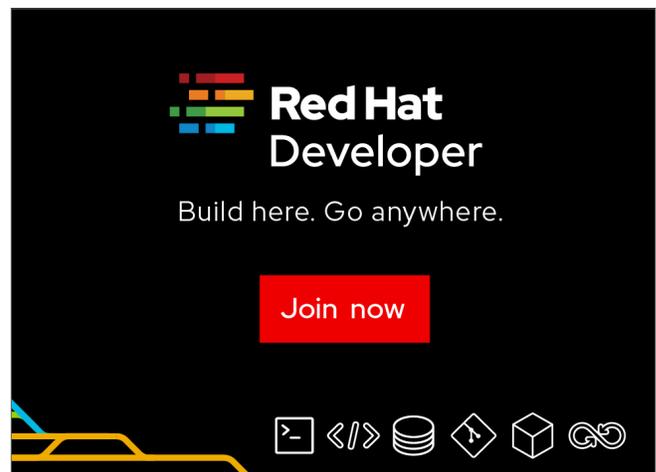
## GETTING STARTED

To create a Quarkus service, you just need to run the next Maven goal into an empty directory:

```
mvn io.quarkus:quarkus-maven-plugin:1.13.1.Final:create \
  \
  -DprojectGroupId=org.acme \
  -DprojectArtifactId=hello-world \
  -DclassName="org.acme.quickstart.GreetingResource" \
  -Dpath="/hello"
```

## LIVE RELOAD

Quarkus applications come with a live reload feature that allows the developer to make changes to their source code, which will be directly reflected in the deployed code without having to recompile or repack the source code.

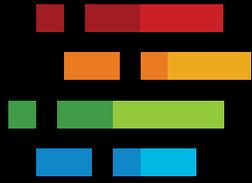


 **Red Hat**  
**Developer**

Build here. Go anywhere.

[Join now](#)

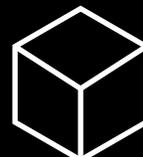
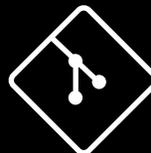
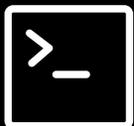
     



# Red Hat Developer

Build here.  
Go anywhere.

Join now



```
./mvnw compile quarkus:dev
```

This service is started locally and you can use a `curl` command on the `http://localhost:8080/hello` URL to get a response from the service.

You can make any changes in the `org.acme.quickstart.GreetingResource` source class, (i.e., change the return value from `hello` to `aloha`), and you can execute a `curl` command on `http://localhost:8080/hello` URL and get the updated response without redeploying the service.

## CONFIGURATION

Quarkus uses the Eclipse MicroProfile Configuration spec as its configuration mechanism. Quarkus applications are fully configured using a single configuration file, which is located at `src/main/resources/application.properties`. You can create a custom property using the following code:

```
greetings.message=Hello World!!
```

Then you can inject the configuration value into any class field:

```
@ConfigProperty(name = "greetings.message")
String message;
```

## PROFILES

Quarkus has three different default profiles that can be set by starting the property with %:

- `dev` when running in `quarkus:dev` mode
- `test` when running tests
- `prod` (the default one)

```
quarkus.http.port=9090
%dev.quarkus.http.port=8181
```

You can also create custom profiles by either setting the active profile as a system property (`quarkus.profile`) or as an environment variable (`QUARKUS_PROFILE`). For example, `Dquarkus.profile=staging`:

```
%staging.quarkus.http.port=9999
```

Configuration values can be overridden at runtime (in decreasing priority):

- Using system properties (`-Dquarkus.http.port`)
- Using environment variables (`QUARKUS_HTTP_PORT`)

- Creating an environment file named `.env` in the current working directory (`GREETING_MESSAGE=Namaste`)
- Creating an external config directory under the current working directory (`config/application.properties`)
- Creating an `application.properties` file in the `src/main/resources` directory within the project

You can set additional configuration files by using the `smallrye.config.locations` property.

## KEY COMPONENTS

### JAX-RS

Quarkus uses the JAX-RS spec for implementing RESTful web services, as shown below:

```
@Path("/developer")
public class DeveloperResource {

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Developer> getAllDevelopers() {}

    @POST
    @Produces(MediaType.APPLICATION_JSON)
    public Response createDeveloper(Developer developer) {}

    @DELETE
    @Path("/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response deleteDeveloper(@PathParam("id")
    Long id) {}

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    @Path("/search")
    public Response searchDeveloper(@QueryParam("skills") String skills) {}
}
```

### JSON MARSHALLING AND UNMARSHALLING

To marshal and unmarshal Java objects to JSON format, you need to register either the JSON-B or Jackson extension:

```
./mvnw quarkus:add-extension -Dextensions="resteasy-jsonb"
```

Or:

```
./mvnw quarkus:add-extension -Dextensions="resteasy-jackson"
```

## MICROPROFILE SPEC FOR CLOUD (KUBERNETES)-NATIVE APPLICATIONS

### RESTFUL WEB CLIENT

Quarkus integrates with MicroProfile REST clients to invoke other RESTful web services.

```
./mvnw quarkus:add-extension -Dextensions="resteasy-  
jsonb, rest-client-jsonb"
```

Below is the code for creating an interface for the RESTful web endpoint accessing <http://worldclockapi.com/api/json/cet/now>:

```
package org.acme.greetings;  
  
@Path("/api")  
@RegisterRestClient  
public interface WorldClockService {  
    @GET  
    @Path("/json/{where}/now")  
    @Produces(MediaType.APPLICATION_JSON)  
    WorldClock getNow(@PathParam("where") String  
        where);  
}
```

Then you need to configure the URL of the service in `application.properties` by using the fully qualified name of the class:

```
org.acme.greetings.WorldClockService/mp-rest/  
url=http://worldclockapi.com
```

Last, you need to inject the interface into the classes that require this client:

```
@Inject  
@RestClient  
WorldClockService worldClockService;
```

**Tip:** If the invocation happens within JAX-RS, you can propagate the headers' value from the incoming request to the outgoing response by specifying them in the `org.eclipse.microprofile.rest.client.propagateHeaders` parameter (i.e., `org.eclipse.microprofile.rest.client.propagateHeaders=Authorization`).

The REST client also supports asynchronous calls by returning `CompletionStage<WorldClock>` or `Uni<WorldClock>` (requires the `quarkus-rest-client-mutiny` extension) instead of the POJO directly.

### FAULT TOLERANCE

It is really important to build fault-tolerant microservices when they are deployed in Kubernetes where all communication happens within the network. Quarkus integrates with the MicroProfile fault-tolerance spec, which uses a CDI interceptor and can be

used in several elements, such as a CDI bean, JAX-RS resource, or MicroProfile REST Client.

```
./mvnw  
quarkus:add-extension -Dextensions="smallrye-fault-  
tolerance"
```

Possible strategies are:

OPERATION	PROPERTIES	ANNOTATION
Retry Policy	@Retry	maxRetries, delay, delayUnit, maxDuration, durationUnit, jitter, jitterDelayUnit, retryOn, abortOn
Fallback Action	@Fallback	fallbackMethod
Timeout	@Timeout	unit
Circuit Breaker	@CircuitBreaker	failOn, skipOn, delay, delayUnit, requestVolumeThreshold, failureRatio, successThreshold
Bulkhead — Thread Pool/ Semaphore	@Bulkhead	waitingTaskQueue (only valid with @Asynchronous semaphore mode)

### AUTOMATIC RETRIES

An automatic retry is executed when the `getNow` method throws an exception.

```
@Retry(maxRetries = 1)  
@Fallback(fallbackMethod = "fallbackMethod")  
WorldClock getNow(){  
  
    public String fallbackMethod() {  
        return WorldClock.now();  
    }  
}
```

The `fallbackMethod` must have the same parameters and return type as the annotated method, as well as an additional `ExecutionContext` parameter.

Fallback logic can be implemented in a class instead of a method, as shown below:

```
public class RecoverFallback implements  
    FallbackHandler<WorldClock> {  
    @Override  
    public WorldClock handle(ExecutionContext context) {  
    }  
}
```

This type of logic can then be set in the annotation as the value `@Fallback(RecoverFallback.class)`.

## CIRCUIT BREAKER

Circuit breaker is used to detect failures and encapsulate the logic, preventing a failure to occur repeatedly.

```
@CircuitBreaker(requestVolumeThreshold = 4,
failureRatio = 0.75, delay = 1000)

@Fallback(fallbackMethod = "fallbackMethod")
WorldClock getNow() {}
```

If three (4 x 0.75) failures occur among the rolling window of four consecutive invocations, the circuit is opened for 1000 milliseconds and then returns to a half open state.

## BULKHEAD

Bulkhead pattern limits the number of concurrent calls to a component.

```
@Bulkhead(5)
@Retry(maxRetries = 4, delay = 1000, retryOn =
BulkheadException.class)
WorldClock getNow() {}
```

## CONFIGURATION

You can override annotation parameters via the configuration file using the following property:

```
[classname/methodname/]annotation/parameter:
# Method scope
org.acme.greetings.WorldClockService/getNow/Retry/
maxDuration=30
# Class scope
org.acme.greetings.WorldClockService/Retry/
maxDuration=3000
# Global
Retry/maxDuration=3000
```

## HEALTH CHECKS

Kubernetes relies on the concept of liveness and readiness probes to monitor the state of the pod and decide if it should be restarted or, for example, if it should start accepting traffic. Quarkus integrates with the MicroProfile Health spec to provide health checks to Kubernetes.

```
./mvnw quarkus:add-extension -Dextensions="smallrye-
health"
```

To create a custom health check, you need to implement the `HealthCheck` interface and annotate the class with `@Readiness` and/or `@Liveness`.

```
@Liveness
@ApplicationScoped
public class DatabaseHealthCheck implements
HealthCheck {
    @Override
    public HealthCheckResponse call() {
        HealthCheckResponseBuilder responseBuilder =
            HealthCheckResponse.named("Database conn");

        try {
            checkDatabaseConnection();
            responseBuilder.withData("connection",
                true);
            responseBuilder.up();
        }
        catch (IOException e) {
            // cannot access the database
            responseBuilder.down()
                .withData("error", e.getMessage());
        }
        return responseBuilder.build();
    }
}
```

Below is the code for accessing the `/q/health` endpoint to get a JSON document describing the status of the application:

```
{
  "status": "UP",
  "checks": [
    {
      "name": "Database conn",
      "status": "UP",
      "data": {
        "connection": true
      }
    }
  ]
}
```

The status code is 200 OK if the global status is UP and 503 Service Unavailable if the status is DOWN. The `/q/health/ready` and `/q/health/live` endpoints can be used to get results individually from readiness or liveness probes.

Since health checks are CDI beans, you can create health checks directly in a class:

```
import javax.enterprise.inject.Produces;
import io.smallrye.health.HealthStatus;

public class ApplicationHealthChecks {

    @Liveness
    @ApplicationScoped
    @Produces
```

*Continued on next page.*

```
public HealthCheck liveCheck() {
    return HealthStatus.up("App");
}

@Readiness
@ApplicationScoped
@Produces
public HealthCheck dbHealthCheck() {
    return HealthStatus.state("db",
this::isDbUpAndRunning);
}

private boolean isDbUpAndRunning() {}
}
```

Some extensions provide readiness probes by default. To mention some of them: DataSource, Kafka, Kafka-Streams, MongoDB, Neo4J, Artemis, Vault, gRPC, Cassandra, and Redis.

## METRICS

Quarkus can utilize the Micrometer metrics library for runtime and application metrics.

```
./mvnw quarkus:add-extension -Dextensions="micrometer-registry-prometheus"
```

With the Prometheus registry extension added, the generated output format follows the Prometheus format.

By default, runtime metrics are generated automatically (i.e., HTTP server connections, JVM memory, etc.), but application metrics can be generated too.

You can use `io.micrometer.core.instrument.MeterRegistry` to register metrics:

```
private final MeterRegistry registry;
public PrimeNumberResource(MeterRegistry registry) {
    this.registry = registry;
    registry.gauge("prime.number.max", this,
        PrimeNumberResource::highestObservedPrimeNumber);
}
}
```

Micrometer uses `MeterFilter` instances to customize the metrics emitted by `MeterRegistry` instances. Any `MeterFilter` CDI beans are detected and used when initializing `MeterRegistry` instances.

```
@Singleton
public class CustomConfiguration {
    @Produces
    @Singleton
    public MeterFilter configureAllRegistries() {
        return MeterFilter.commonTags(Arrays.asList(
            Tag.of("env", deploymentEnv)));
    }
}
```

Metrics are accessible at `/q/metrics/`.

## TRACING

Quarkus uses OpenTracing (the MicroProfile OpenTracing spec) to provide distributed tracing for services distributed across a Kubernetes cluster.

```
./mvnw quarkus:add-extension
-Dextensions="smallrye-opentracing"
```

The below code is used for tracing configuration:

```
quarkus.jaeger.service-name=myservice
quarkus.jaeger.sampler-type=const
quarkus.jaeger.sampler-param=1
quarkus.jaeger.endpoint=http://localhost:14268/api/traces
```

By default, requests sent to any endpoint will be traced without any code changes being required. You can customize traces by injecting the `Tracer` class:

```
@Inject
Tracer tracer;
tracer.activeSpan().setBaggageItem("key", "value");
```

If you want to disable tracing at the class or method level, you can use the `@Traced` annotation.

Quarkus provides additional tracers:

- **JDBC Tracer:** Adds a span for each JDBC query.
  - You need to register the `io.opentracing.contrib:opentracing-jdbc` dependency and configure the `datasource` url parameter with `jdbc:tracing:postgresql://localhost:5432/mydatabase` and the driver parameter with `io.opentracing.contrib.jdbc.TracingDriver`.
- **Kafka Tracer:** Adds a span for each message sent to, or received from, a Kafka topic.
  - You need to register the `io.opentracing.contrib:opentracing-kafka-client` dependency and configure incoming and outgoing interceptors with `io.opentracing.contrib.kafka.TracingProducerInterceptor` and `io.opentracing.contrib.kafka.TracingConsumerInterceptor`.

## SECURITY AND JWT

In microservices architectures, using stateless services is one of the goals you'd like to achieve. For security and, more specifically, the authorization process, Quarkus integrates with the MicroProfile JWT RBAC spec.

```
./mvnw quarkus:add-extension -Dextensions="smallrye-jwt"
```

To send a token to the server-side, you should use the Authorization header:

```
curl -H \"Authorization: Bearer eyJraWQiOi... \".
```

The minimum JWT required claims are: `typ`, `alg`, `kid`, `iss`, `sub`, `exp`, `iat`, `jti`, `upn`, and `groups`.

Quarkus ensures that the given token is valid (i.e., not manipulated, not expired, from the correct issuer, etc.). JWT tokens can be injected into a `JsonWebToken` class in your code or using `@Claim` if a specific claim is required.

```
@Inject
JsonWebToken jwt;
@Inject
@Claim(standard = Claims.preferred_username)
String name;
```

To inject claim values, the bean must be `@RequestScoped` CDI scoped. If you need to inject claim values into the scope with a lifetime greater than `@RequestScoped`, then you need to use the `javax.enterprise.inject.Instance` interface.

An example of the minimal configuration parameters is:

```
mp.jwt.verify.publickey.location=META-INF/resources/
publicKey.pem

mp.jwt.verify.issuer=https://quarkus.io/using-jwt-rbac
```

PARAMETER	DEFAULT	DESCRIPTION
<code>quarkus.smallrye-jwt.enabled</code>	<code>true</code>	Determine if the JWT extension is enabled.
<code>quarkus.smallrye-jwt.rsa-sig-provider</code>	<code>SunRsaSign</code>	The name of the <code>java.security.Provider</code> that supports SHA256withRSA signatures.
<code>mp.jwt.verify.publickey</code>		Public Key text itself to be supplied as a string.
<code>mp.jwt.verify.publickey.location</code>		Relative path or URL of a public key.
<code>mp.jwt.verify.issuer</code>		<code>iss</code> accepted as valid.

The JWT groups claim is directly mapped to roles used in security annotations by using security annotations (i.e., `@RolesAllowed("Subscriber")`)

## DOCKER AND NATIVE

Quarkus's scaffolding project comes with Dockerfiles to generate Docker images for using Quarkus in JVM mode or in native mode.

### JVM MODE

In this mode, no native compilation is required and you simply run your application via a traditional jar.

```
./mvnw clean package
docker build -f src/main/docker/Dockerfile.jvm -t
quarkus/gettingstarted .
```

### NATIVE MODE

You can build a native image by using GraalVM. Quarkus can generate a native executable of your application to be containerized into Docker without having to install GraalVM in your local machine.

```
./mvnw package -Pnative -Dnative-image.dockerbuild=true
docker build -f src/main/docker/Dockerfile.native -t
quarkus/gettingstarted .
```

## CONTAINER IMAGES

You can leverage Quarkus to generate and release container images.

Three methods are supported based on the registered extension:

- Jib ([GoogleContainerTools/jib: Build container images for your Java applications.](#))
  - `./mvnw quarkus:add-extension -Dextensions="quarkus-container-image-jib"`
- Docker
  - `./mvnw quarkus:add-extension -Dextensions="quarkus-container-image-docker"`
- S2I ([openshift/source-to-image: A tool for building artifacts from source and injecting into container images](#))
  - `./mvnw quarkus:add-extension -Dextensions="quarkus-container-image-s2i"`

PROPERTY	DEFAULT	DESCRIPTION
<code>quarkus.container-image.group</code>	<code>\${user.name}</code>	The group/repository of the image.
<code>quarkus.container-image.name</code>	The application name	The name of the image.
<code>quarkus.container-image.tag</code>	The application version	The tag of the image.

*Continued on next page.*

<code>quarkus.container-image.additional-tags</code>		Additional tags of the container image.
<code>quarkus.container-image.registry</code>	<code>docker.io</code>	The registry to use for pushing.
<code>quarkus.container-image.username</code>		The username to access the registry.
<code>quarkus.container-image.password</code>		The password to access the registry.
<code>quarkus.container-image.insecure</code>	<code>false</code>	Flag to allow insecure registries.
<code>quarkus.container-image.build</code>	<code>false</code>	Boolean to set if the image should be built.
<code>quarkus.container-image.push</code>	<code>false</code>	Boolean to set if the image should be pushed.

To build and push a container image using the Quarkus extension, you need to run the following command:

```
./mvnw package -Dquarkus.container-image.push=true
```

**Tip:** You can use native flags and container image tags together to publish a container image containing a native image.

## KUBERNETES AND QUARKUS INTEGRATION

Quarkus can use the [Dekorator project](#) to generate Kubernetes resources.

```
./mvnw quarkus:add-extension -Dextensions="quarkus-kubernetes"
```

By running `./mvnw package`, the Kubernetes resources are created at the `target/wiring-classes/META-INF/kubernetes` directory. Then the generated resources are integrated with the MicroProfile Health and Metrics annotations.

You can customize the generated resources by setting new values in `application.properties`:

```
quarkus.kubernetes.namespace=mynamespace

quarkus.kubernetes.replicas=3

quarkus.kubernetes.labels.foo=bar

quarkus.kubernetes.annotations.foo=bar

quarkus.kubernetes.readiness-probe.period-seconds=45
```

*Continued on next column.*

```
quarkus.kubernetes.mounts.github-token.path=/
deployment/github
quarkus.kubernetes.mounts.github-token.read-only=true

quarkus.kubernetes.secret-volumes.github-token.volume-
name=github-token
quarkus.kubernetes.secret-volumes.github-token.secret-
name=greeting-security
quarkus.kubernetes.secret-volumes.github-token.default-
mode=420

quarkus.kubernetes.config-map-volumes.github-token.
config-map-name=my-secret

quarkus.kubernetes.expose=true

quarkus.kubernetes.env.vars.my-env-var=foobar
quarkus.kubernetes.env.configmaps=my-config-
map,another-config-map
quarkus.kubernetes.env.secrets=my-secret,my-other-
secret

quarkus.kubernetes.resources.requests.memory=64Mi
quarkus.kubernetes.resources.limits.cpu=1000m
```

Resources can be added under the `src/main/kubernetes` directory.

PARAMETER	DEFAULT	DESCRIPTION
<code>quarkus.kubernetes.deploy</code>	<code>false</code>	Generates and deploys the resources to Kubernetes.
<code>quarkus.kubernetes.deployment-target</code>	Kubernetes	Generates resources for kubernetes, openshift, or knative.

To generate a container image, push it to a registry, and deploy the application, you can run the following command:

```
./mvnw clean package -Dquarkus.kubernetes.deploy=true
```

## REMOTE LIVE RELOAD

Live reload also works with remote instances (even when deployed inside a Kubernetes cluster) if it is configured properly in `src/main/resources/application.properties`:

```
# Mutable Jar configurations
quarkus.package.type=mutable-jar
quarkus.live-reload.password=changeit

quarkus.container-image.build=true
quarkus.kubernetes-client.trust-certs=true
quarkus.kubernetes.expose=true
quarkus.kubernetes.env-vars.quarkus-launch-devmode.
value=true
```

Notice the application is configured to start in dev mode — even

in production mode — by setting the `quarkus-launch-devmode` environment variable to `true`.

With application deployed in the cluster, get the public URL to access the service and set it in the configuration file:

```
quarkus.live-reload.url=http://YOUR_APP_ROUTE_URL
```

Start the application in remote-dev mode:

```
./mvnw quarkus:remote-dev
```

Now any changes made locally will automatically be reloaded by the application in the Kubernetes cluster.

## CONCLUSION

Java has evolved since its introduction in 1995, but more importantly, the environments hosting applications have evolved as well. Some trade-offs made in Java's design at the onset affect how Java is perceived today. Things that were important in those days aren't as important anymore. Quarkus was invented given the challenges and problems of today and aims to solve those challenges without forcing developers to learn an entirely new programming language.

This Refcard has shown how to use some of the most common extensions within the Quarkus ecosystem. Please visit <https://code.quarkus.io> for a list of all extensions as well as <https://quarkus.io/guides/> for further information on building Quarkus applications.



### WRITTEN BY ALEX SOTO,

*DIRECTOR OF DEVELOPER EXPERIENCE, RED HAT*

Alex is a Director of Developer Experience at Red Hat. He is passionate about Java and software automation, and he believes in the open source software model.

Alex is the creator of the NoSQLUnit project, a member of the JSR374 (Java API for JSON Processing) Expert Group, the co-author of the *Testing Java Microservices* book from Manning, and a contributor to several open source projects. A Java Champion since 2017, international speaker and teacher at Salle URL University, he has talked about new testing techniques for microservices and continuous delivery in the 21st century.



DZone, a Devada Media Property, is the resource software developers, engineers, and architects turn to time and again to learn new skills, solve software development problems, and share their expertise. Every day, hundreds of thousands of developers come to DZone to read about the latest technologies, methodologies, and best practices. That makes DZone the ideal place for developer marketers to build product and brand awareness and drive sales. DZone clients include some of the most innovative technology and tech-enabled companies in the world including Red Hat, Cloud Elements, Sensu, and Sauce Labs.

Devada, Inc.  
600 Park Offices Drive  
Suite 150  
Research Triangle Park, NC 27709  
888.678.0399 919.678.0300

Copyright © 2020 Devada, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means of electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.