

# Quarkus - Sending emails

This guide demonstrates how your Quarkus application can send emails using an SMTP server.

## Prerequisites

To complete this guide, you need:

- less than 15 minutes
- The SMTP hostname, port and credentials, and an email address
- an IDE
- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- Apache Maven 3.6.3
- GraalVM installed if you want to run in native mode.

## Architecture

In this guide, we are going to see how you can send emails from a Quarkus application. It covers simple emails, attachments, inlined attachments, the reactive and imperative APIs...

## Creating the Maven Project

Create a new project with the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.8.0.Final:create \
  -DprojectId=org.acme \
  -DprojectArtifactId=sending-email-quickstart \
  -Dextensions="mailer"
cd sending-email-quickstart
```

If you already have an existing project, add the `mailer` extension:

```
./mvnw quarkus:add-extensions -Dextensions="mailer"
```

This will add the following to your `pom.xml`:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-mailer</artifactId>
</dependency>
```

## Configuring the mailer

The Quarkus mailer is using SMTP. In the `src/main/resources/application.properties` file, you need to configure the host, port, username, password as well as the other configuration aspect. Note that the password can also be configured using system properties and environment variables.

Here is an example using *sendgrid*:

```
quarkus.mailer.from=test@quarkus.io
quarkus.mailer.host=smtp.sendgrid.net
quarkus.mailer.port=465
quarkus.mailer.ssl=true
quarkus.mailer.username=...
quarkus.mailer.password=...
quarkus.mailer.mock=false
```

It is recommended to encrypt any sensitive data, such as the `quarkus.mailer.password`. One approach is to save the value into a secure store like HashiCorp Vault, and refer to it from the configuration. Assuming for instance that Vault contains key `mail-password` at path `myapps/myapp/myconfig`, then the mailer extension can be simply configured as:



```
...
# path within the kv secret engine where is located the
application sensitive configuration
quarkus.vault.secret-config-kv-
path=myapps/myapp/myconfig
...
quarkus.mailer.password=${mail-password}
```

Please note that the password value is evaluated only once, at startup time. If `mail-password` was changed in Vault, the only way to get the new value would be to restart the application.



For more information about the Mailer extension configuration please refer to the [Configuration Reference](#).

# Sending simple emails

In a JAX-RS resource, or in a bean, you can inject the mailer as follows:

```
@Inject
Mailer mailer;

@Inject
ReactiveMailer reactiveMailer;
```

There are 2 APIs:

- `io.quarkus.mailer.Mailer` provides the imperative (blocking and synchronous) API;
- `io.quarkus.mailer.reactive.ReactiveMailer` provides the reactive (non-blocking and asynchronous) API



The two APIs are equivalent feature-wise. Actually the `Mailer` implementation is built on top of the `ReactiveMailer` implementation.



*Deprecation*

`io.quarkus.mailer.ReactiveMailer` is deprecated in favor of `io.quarkus.mailer.reactive.ReactiveMailer`.



*Mutiny*

The reactive mailer uses Mutiny reactive types, if you're not familiar with them, read the [Getting Started with Reactive guide](#) first.

To send a simple email, proceed as follows:

```
// Imperative API:
mailer.send(Mail.withText("to@acme.org", "A simple email from
quarkus", "This is my body.));
// Reactive API:
Uni<Void> stage = reactiveMailer.send(Mail.withText("to@acme.org",
"A reactive email from quarkus", "This is my body.));
```

For example, you can use the `Mailer` in a JAX-RS endpoint as follows:

```

@GET
@Path("/simple")
public Response sendASimpleEmail() {
    mailer.send(Mail.withText("to@acme.org", "A simple email from
quarkus", "This is my body"));
    return Response.accepted().build();
}

@GET
@Path("/async")
public CompletionStage<Response> sendASimpleEmailAsync() {
    return reactiveMailer.send(
        Mail.withText("to@acme.org", "A reactive email from
quarkus", "This is my body"))
        .subscribeAsCompletionStage()
        .thenApply(x -> Response.accepted().build());
}

```



With the `quarkus-resteasy-mutiny` extension, you can return an instance of `Uni` directly.

With such a JAX-RS resource, you can check that everything is working with:

```

curl http://localhost:8080/simple
curl http://localhost:8080/async

```

You can create new `io.quarkus.mailer.Mail` instances from the constructor or from the `Mail.withText` and `Mail.withHtml` helper methods. The `Mail` instance lets you add recipients (to, cc, or bcc), set the subject, headers, sender (from) address...

You can also send several `Mail` objects in one call:

```

mailer.send(mail1, mail2, mail3);

```

## Sending attachments

To send attachment, just use the `addAttachment` methods on the `io.quarkus.mailer.Mail` instance:

```

@GET
@Path("/attachment")
public Response sendEmailWithAttachment() {
    mailer.send(Mail.withText("to@acme.org", "An email from quarkus
with attachment",
        "This is my body")
        .addAttachment("my-file.txt",
            "content of my file".getBytes(), "text/plain"));
    return Response.accepted().build();
}

```

Attachments can be created from raw bytes (as shown in the snippet) or files.

## Sending HTML emails with inlined attachments

When sending HTML email, you can add inlined attachments. For example, you can send an image with your email, and this image will be displayed in the mail content. If you put the image file into resources folder, you should specify the full path to the file. "e.g." "META-INF/resources/quarkus-logo.png" otherwise quarkus will lookup in the root folder of the project

```

@GET
@Path("/html")
public Response sendingHTML() {
    String body = "<strong>Hello!</strong>" + "\n" +
        "<p>Here is an image for you: <img src=\"cid:my-
image@quarkus.io\"/></p>" +
        "<p>Regards</p>";
    mailer.send(Mail.withHtml("to@acme.org", "An email in HTML",
body)
        .addInlineAttachment("quarkus-logo.png",
            new File("quarkus-logo.png"),
            "image/png", "<my-image@quarkus.io>"));
    return Response.accepted().build();
}

```

Note the *content-id* format and reference. By spec, when you create the inline attachment, the content-id must be structured as follows: `<id@domain>`. If you don't wrap your content-id between `<>`, it is automatically wrapped for you. When you want to reference your attachment, for instance in the `src` attribute, use `cid:id@domain` (without the `<` and `>`).

## Message Body Based on Qute Templates

It's also possible to inject a mail template, where the message body is created automatically using [Qute templates](#).

```

@Path("")
public class MailingResource {

    @CheckedTemplate
    class Templates {
        public static native MailTemplateInstance hello(String
name); ①
    }

    @GET
    @Path("/mail")
    public CompletionStage<Response> send() {
        // the template looks like: Hello {name}! ②
        return Templates.hello("John")
            .to("to@acme.org") ③
            .subject("Hello from Qute template")
            .send() ④
            .subscribeAsCompletionStage()
            .thenApply(x -> Response.accepted().build());
    }
}

```

- ① By convention, the enclosing class name and method names are used to locate the template. In this particular case, we will use the `MailingResource/hello.html` and `MailingResource/hello.txt` templates to create the message body.
- ② Set the data used in the template.
- ③ Create a mail template instance and set the recipient.
- ④ `MailTemplate.send()` triggers the rendering and, once finished, sends the e-mail via a `Mailer` instance.



Injected mail templates are validated during build. If there is no matching template in `src/main/resources/templates` the build fails.

You can also do this without type-safe templates:

```

@Inject
MailTemplate hello; ①

@GET
@Path("/mail")
public CompletionStage<Response> send() {
    return hello.to("to@acme.org") ②
        .subject("Hello from Qute template")
        // the template looks like: Hello {name}!
        .data("name", "John") ③
        .send() ④
        .subscribeAsCompletionStage()
        .thenApply(x -> Response.accepted().build());
}

```

- ① If there is no `@ResourcePath` qualifier provided, the field name is used to locate the template. In this particular case, we will use the `hello.html` and `hello.txt` templates to create the message body.
- ② Create a mail template instance and set the recipient.
- ③ Set the data used in the template.
- ④ `MailTemplate.send()` triggers the rendering and, once finished, sends the e-mail via a `Mailer` instance.



Injected mail templates are validated during build. If there is no matching template in `src/main/resources/templates` the build fails.

## Testing email sending

Because it is very inconvenient to send emails during development and testing, you can set the `quarkus.mailer.mock` boolean configuration to `true` to not actually send emails but print them on stdout and collect them in a `MockMailbox` bean instead. This is the default if you are running Quarkus in `DEV` or `TEST` mode.

You can then write tests to verify that your emails were sent, for example, by a REST endpoint:

```

@QuarkusTest
class MailTest {

    private static final String T0 = "foo@quarkus.io";

    @Inject
    MockMailbox mailbox;

    @BeforeEach
    void init() {
        mailbox.clear();
    }

    @Test
    void testTextMail() throws MessagingException, IOException {
        // call a REST endpoint that sends email
        given()
        .when()
        .get("/send-email")
        .then()
            .statusCode(202)
            .body(is("OK"));

        // verify that it was sent
        List<Mail> sent = mailbox.getMessagesSentTo(T0);
        assertThat(sent).hasSize(1);
        Mail actual = sent.get(0);
        assertThat(actual.getText()).contains("Wake up!");
        assertThat(actual.getSubject()).isEqualTo("Alarm!");

        assertThat(mailbox.getTotalMessagesSent()).isEqualTo(6);
    }
}

```

## Gmail specific configuration

If you want to use the Gmail SMTP server, first create a dedicated password in [Google Account > Security > App passwords](#) or go to <https://myaccount.google.com/apppasswords>.

When done, you can configure your Quarkus application by adding the following properties to your `application.properties`:

With TLS:

```
quarkus.mailer.auth-methods=DIGEST-MD5 CRAM-SHA256 CRAM-SHA1 CRAM-
MD5 PLAIN LOGIN
quarkus.mailer.from=YOUREMAIL@gmail.com
quarkus.mailer.host=smtp.gmail.com
quarkus.mailer.port=587
quarkus.mailer.start-tls=REQUIRED
quarkus.mailer.username=YOUREMAIL@gmail.com
quarkus.mailer.password=YOURGENERATEDAPPLICATIONPASSWORD
```

Or with SSL:

```
quarkus.mailer.auth-methods=DIGEST-MD5 CRAM-SHA256 CRAM-SHA1 CRAM-
MD5 PLAIN LOGIN
quarkus.mailer.from=YOUREMAIL@gmail.com
quarkus.mailer.host=smtp.gmail.com
quarkus.mailer.port=465
quarkus.mailer.ssl=true
quarkus.mailer.username=YOUREMAIL@gmail.com
quarkus.mailer.password=YOURGENERATEDAPPLICATIONPASSWORD
```



The `quarkus.mailer.auth-methods` configuration option is needed for the Quarkus mailer to support password authentication with Gmail. By default both the mailer and Gmail default to `XOAUTH2` which requires registering an application, getting tokens, etc.

## Using SSL with native executables

Note that if you enable SSL for the mailer and you want to build a native executable, you will need to enable the SSL support. Please refer to the [Using SSL With Native Executables](#) guide for more information.

## Using the underlying Vert.x Mail Client

The Quarkus Mailer is implemented on top of the [Vert.x Mail Client](#), providing an asynchronous and non-blocking way to send emails. If you need fine control on how the mail is sent, for instance if you need to retrieve the message ids, you can inject the underlying client, and use it directly:

```
@Inject MailClient client;
```

Three API flavors are exposed:

- the Mutiny client (`io.vertx.mutiny.ext.mail.MailClient`)
- the Axle client (`io.vertx.axle.ext.mail.MailClient`), using `CompletionStage` and

Reactive Streams **Publisher** - deprecated, it is recommended to switch to the Mutiny client

- the RX Java 2 client (`io.vertx.reactivex.ext.mail.MailClient`) - deprecated, it is recommended to switch to the Mutiny client
- the bare client (`io.vertx.ext.mail.MailClient`)

Check the [Using Vert.x guide](#) for further details about these different APIs and how to select the most suitable for you.

The retrieved **MailClient** is configured using the configuration key presented above. You can also create your own instance, and pass your own configuration.

## Conclusion

This guide has shown how you can send emails from a Quarkus application. The *mailer* extension works in JVM and native mode.

## Mailer Configuration Reference

 Configuration property fixed at build time - All other configuration properties are overridable at runtime

Configuration property	Type	Default
<code>quarkus.mailer.from</code> Configure the default <b>from</b> attribute. It's the sender email address.	string	
<code>quarkus.mailer.mock</code> Enables the mock mode, not sending emails. The content of the emails is printed on the console. Disabled by default on PROD, enabled by default on DEV and TEST modes.	boolean	
<code>quarkus.mailer.bounce-address</code> Configures the default bounce email address.	string	
<code>quarkus.mailer.host</code> The SMTP host name.	string	<b>localhost</b>
<code>quarkus.mailer.port</code> The SMTP port.	int	

<code>quarkus.mailer.username</code>		
The username.	string	
<code>quarkus.mailer.password</code>		
The password.	string	
<code>quarkus.mailer.ssl</code>		
Enables or disables the SSL on connect. <code>false</code> by default.	boolean	<code>false</code>
<code>quarkus.mailer.trust-all</code>		
Set whether to trust all certificates on ssl connect the option is also applied to <code>STARTTLS</code> operation. <code>false</code> by default.	boolean	<code>false</code>
<code>quarkus.mailer.max-pool-size</code>		
Configures the maximum allowed number of open connections to the mail server If not set the default is <code>10</code> .	int	
<code>quarkus.mailer.own-host-name</code>		
The hostname to be used for HELO/EHLO and the Message-ID	string	
<code>quarkus.mailer.keep-alive</code>		
Set if connection pool is enabled, <code>true</code> by default. If the connection pooling is disabled, the max number of sockets is enforced nevertheless.	boolean	<code>true</code>
<code>quarkus.mailer.disable-esmtp</code>		
Disable ESMTP. <code>false</code> by default. The RFC-1869 states that clients should always attempt <code>EHLO</code> as first command to determine if ESMTP is supported, if this returns an error code, <code>HELO</code> is tried to use the <b>regular</b> SMTP command.	boolean	<code>false</code>
<code>quarkus.mailer.start-tls</code>		
Set the TLS security mode for the connection. Either <code>DISABLED</code> , <code>OPTIONAL</code> or <code>REQUIRED</code> .	string	
<code>quarkus.mailer.login</code>		
Set the login mode for the connection. Either <code>DISABLED</code> , <code>OPTIONAL</code> or <code>REQUIRED</code>	string	

<p><code>quarkus.mailer.auth-methods</code></p> <p>Set the allowed auth methods. If defined, only these methods will be used, if the server supports them.</p>	string	
<p><code>quarkus.mailer.key-store</code></p> <p>Set the key store.</p>	string	
<p><code>quarkus.mailer.key-store-password</code></p> <p>Set the key store password.</p>	string	