Qute Templating Engine

Qute is a templating engine designed specifically to meet the Quarkus needs. The usage of reflection is minimized to reduce the size of native images. The API combines both the imperative and the non-blocking reactive style of coding. In the development mode, all files located in `src/main/resources/templates` are watched for changes and modifications are immediately visible. Furthermore, we try to detect most of the template problems at build time. In this guide, you will learn how to easily render templates in your application.

> This technology is considered experimental.
>
> In *experimental* mode, early feedback is requested to mature the idea. There is no guarantee of stability nor long term presence in the platform until the solution matures. Feedback is welcome on our mailing list or as issues in our GitHub issue tracker.
>
> For a full list of possible extension statuses, check our FAQ entry.

Hello World with JAX-RS

If you want to use Qute in your JAX-RS application, you need to add the `quarkus-qute-resteasy` extension first. In your `pom.xml` file, add:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-resteasy-qute</artifactId>
</dependency>
```

We'll start with a very simple template:

hello.txt

```
Hello {name}!  ①
```

① `{name}` is a value expression that is evaluated when the template is rendered.

> By default, all files located in the `src/main/resources/templates` directory and its subdirectories are registered as templates. Templates are validated during startup and watched for changes in the development mode.

Now let's inject the "compiled" template in the resource class.

HelloResource.java

```java
package org.acme.quarkus.sample;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.QueryParam;

import io.quarkus.qute.TemplateInstance;
import io.quarkus.qute.Template;

@Path("hello")
public class HelloResource {

    @Inject
    Template hello; ①

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public TemplateInstance get(@QueryParam("name") String name) {
        return hello.data("name", name); ② ③
    }
}
```

① If there is no `@ResourcePath` qualifier provided, the field name is used to locate the template. In this particular case, we're injecting a template with path `templates/hello.txt`.

② `Template.data()` returns a new template instance that can be customized before the actual rendering is triggered. In this case, we put the name value under the key `name`. The data map is accessible during rendering.

③ Note that we don't trigger the rendering - this is done automatically by a special `ContainerResponseFilter` implementation.

If your application is running, you can request the endpoint:

```
$ curl -w "\n" http://localhost:8080/hello?name=Martin
Hello Martin!
```

Type-safe templates

There's an alternate way to declare your templates in your Java code, which relies on the following convention:

- Organise your template files in the `/src/main/resources/templates` directory, by grouping them into one directory per resource class. So, if your `ItemResource` class references two templates `hello` and `goodbye`, place them at

`/src/main/resources/templates/ItemResource/hello.txt` and `/src/main/resources/templates/ItemResource/goodbye.txt`. Grouping templates per resource class makes it easier to navigate to them.

- In each of your resource class, declare a `@CheckedTemplate static class Template {}` class within your resource class.

- Declare one `public static native TemplateInstance method();` per template file for your resource.

- Use those static methods to build your template instances.

Here's the previous example, rewritten using this style:

We'll start with a very simple template:

HelloResource/hello.txt

```
Hello {name}! ①
```

① `{name}` is a value expression that is evaluated when the template is rendered.

Now let's declare and use those templates in the resource class.

HelloResource.java

```java
package org.acme.quarkus.sample;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.QueryParam;

import io.quarkus.qute.TemplateInstance;
import io.quarkus.qute.api.CheckedTemplate;

@Path("hello")
public class HelloResource {

    @CheckedTemplate
    public static class Templates {
        public static native TemplateInstance hello(); ①
    }

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public TemplateInstance get(@QueryParam("name") String name) {
        return Templates.hello().data("name", name); ② ③
    }
}
```

① This declares a template with path `templates/HelloResource/hello.txt`.

② `Templates.hello()` returns a new template instance that can be customized before the actual rendering is triggered. In this case, we put the name value under the key `name`. The data map is accessible during rendering.

③ Note that we don't trigger the rendering - this is done automatically by a special `ContainerResponseFilter` implementation.

> Once you have declared a `@CheckedTemplate` class, we will check that all its methods point to existing templates, so if you try to use a template from your Java code and you forgot to add it, we will let you know at build time :)

Keep in mind this style of declaration allows you to reference templates declared in other resources too:

HelloResource.java

```java
package org.acme.quarkus.sample;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.QueryParam;

import io.quarkus.qute.TemplateInstance;

@Path("goodbye")
public class GoodbyeResource {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public TemplateInstance get(@QueryParam("name") String name) {
        return HelloResource.Templates.hello().data("name", name);
    }
}
```

Toplevel type-safe templates

Naturally, if you want to declare templates at the toplevel, directly in `/src/main/resources/templates/hello.txt`, for example, you can declare them in a toplevel (non-nested) `Templates` class:

```
package org.acme.quarkus.sample;

import io.quarkus.qute.TemplateInstance;
import io.quarkus.qute.Template;
import io.quarkus.qute.api.CheckedTemplate;

@CheckedTemplate
public class Templates {
    public static native TemplateInstance hello();  ①
}
```

① This declares a template with path `templates/hello.txt`.

Template Parameter Declarations

If you declare a **parameter declaration** in a template then Qute attempts to validate all expressions that reference this parameter and if an incorrect expression is found the build fails.

Let's suppose we have a simple class like this:

Item.java

```
public class Item {
    public String name;
    public BigDecimal price;
}
```

And we'd like to render a simple HTML page that contains the item name and price.

Let's start again with the template:

ItemResource/item.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>{item.name}</title>  ①
</head>
<body>
    <h1>{item.name}</h1>
    <div>Price: {item.price}</div>  ②
</body>
</html>
```

① This expression is validated. Try to change the expression to `{item.nonSense}` and the build

should fail.

② This is also validated.

Finally, let's create a resource class with type-safe templates:

ItemResource.java

```java
package org.acme.quarkus.sample;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

import io.quarkus.qute.TemplateInstance;
import io.quarkus.qute.Template;
import io.quarkus.qute.api.CheckedTemplate;

@Path("item")
public class ItemResource {

    @CheckedTemplate
    public static class Templates {
        public static native TemplateInstance item(Item item); ①
    }

    @GET
    @Path("{id}")
    @Produces(MediaType.TEXT_HTML)
    public TemplateInstance get(@PathParam("id") Integer id) {
        return Templates.item(service.findItem(id)); ②
    }
}
```

① Declare a method that gives us a `TemplateInstance` for `templates/ItemResource/item.html` and declare its `Item item` parameter so we can validate the template.

② Make the `Item` object accessible in the template.

Template parameter declaration inside the template itself

Alternatively, you can declare your template parameters in the template file itself.

Let's start again with the template:

item.html

```
{@org.acme.Item item} ①
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>{item.name}</title> ②
</head>
<body>
    <h1>{item.name}</h1>
    <div>Price: {item.price}</div>
</body>
</html>
```

① Optional parameter declaration. Qute attempts to validate all expressions that reference the parameter `item`.

② This expression is validated. Try to change the expression to `{item.nonSense}` and the build should fail.

Finally, let's create a resource class.

ItemResource.java

```java
package org.acme.quarkus.sample;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

import io.quarkus.qute.TemplateInstance;
import io.quarkus.qute.Template;

@Path("item")
public class ItemResource {

    @Inject
    ItemService service;

    @Inject
    Template item; ①

    @GET
    @Path("{id}")
    @Produces(MediaType.TEXT_HTML)
    public TemplateInstance get(@PathParam("id") Integer id) {
        return item.data("item", service.findItem(id)); ②
    }
}
```

① Inject the template with path `templates/item.html`.

② Make the `Item` object accessible in the template.

Template Extension Methods

Template extension methods are used to extend the set of accessible properties of data objects.

Sometimes, you're not in control of the classes that you want to use in your template, and you cannot add methods to them. Template extension methods allows you to declare new method for those classes that will be available from your templates just as if they belonged to the target class.

Let's keep extending on our simple HTML page that contains the item name, price and add a discounted price. The discounted price is sometimes called a "computed property". We will implement a template extension method to render this property easily. Let's update our template:

```html
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>{item.name}</title>
</head>
<body>
    <h1>{item.name}</h1>
    <div>Price: {item.price}</div>
    {#if item.price > 100} ①
    <div>Discounted Price: {item.discountedPrice}</div> ②
    {/if}
</body>
</html>
```

① `if` is a basic control flow section.

② This expression is also validated against the `Item` class and obviously there is no such property declared. However, there is a template extension method declared on the `TemplateExtensions` class - see below.

Finally, let's create a class where we put all our extension methods:

TemplateExtensions.java

```java
package org.acme.quarkus.sample;

import io.quarkus.qute.TemplateExtension;

@TemplateExtension
public class TemplateExtensions {

    public static BigDecimal discountedPrice(Item item) { ①
        return item.price.multiply(new BigDecimal("0.9"));
    }
}
```

① A static template extension method can be used to add "computed properties" to a data class. The class of the first parameter is used to match the base object and the method name is used to match the property name.

> ℹ️ you can place template extension methods in every class if you annotate them with `@TemplateExtension` but we advise to keep them either grouped by target type, or in a single `TemplateExtensions` class by convention.

Rendering Periodic Reports

Templating engine could be also very useful when rendering periodic reports. You'll need to add the `quarkus-scheduler` and `quarkus-qute` extensions first. In your `pom.xml` file, add:

```xml
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-qute</artifactId>
</dependency>
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-scheduler</artifactId>
</dependency>
```

Let's suppose the have a `SampleService` bean whose `get()` method returns a list of samples.

Sample.java

```java
public class Sample {
    public boolean valid;
    public String name;
    public String data;
}
```

The template is simple:

report.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Report {now}</title>
</head>
<body>
    <h1>Report {now}</h1>
    {#for sample in samples} ①
      <h2>{sample.name ?: 'Unknown'}</h2> ②
      <p>
      {#if sample.valid}
        {sample.data}
      {#else}
        <strong>Invalid sample found</strong>.
      {/if}
      </p>
    {/for}
</body>
</html>
```

① The loop section makes it possible to iterate over iterables, maps and streams.

② This value expression is using the elvis operator - if the name is null the default value is used.

ReportGenerator.java

```java
package org.acme.quarkus.sample;

import javax.inject.Inject;

import io.quarkus.qute.Template;
import io.quarkus.qute.api.ResourcePath;
import io.quarkus.scheduler.Scheduled;

public class ReportGenerator {

    @Inject
    SampleService service;

    @ResourcePath("reports/v1/report_01") ①
    Template report;

    @Scheduled(cron="0 30 * * * ?") ②
    void generate() {
        String result = report
            .data("samples", service.get())
            .data("now", java.time.LocalDateTime.now())
            .render(); ③
        // Write the result somewhere...
    }
}
```

① In this case, we use the `@ResourcePath` qualifier to specify the template path: `templates/reports/v1/report_01.html`.

② Use the `@Scheduled` annotation to instruct Quarkus to execute this method on the half hour. For more information see the Scheduler guide.

③ The `TemplateInstance.render()` method triggers rendering. Note that this method blocks the current thread.

Reactive and Asynchronous APIs

Templates can be rendered as a `CompletionStage<String>` (completed with the rendered output asynchronously) or as `Publisher<String>` containing the rendered chunks:

```java
CompletionStage<String> async = template.data("name",
"neo").renderAsync();
Publisher<String> publisher = template.data("name",
"neo").publisher();
```

In the case of a `Publisher`, the template is rendered chunk by chunk following the requests from the

subscriber. The rendering is not started until a subscriber requests it. The returned `Publisher` is an instance of `io.smallrye.mutiny.Multi`.

It is possible to create an instance of `io.smallrye.mutiny.Uni` as follows:

```
Uni<String> uni = Uni.createFrom().completionStage(() ->
template.data("name", "neo").renderAsync());
```

In this case, the rendering only starts once the subscriber requests it.

Qute Reference Guide

To learn more about Qute, please refer to the Qute reference guide.

Qute Configuration Reference

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

Configuration property	Type	Default
🔒 `quarkus.qute.suffixes` The set of suffixes used when attempting to locate a template file. By default, `engine.getTemplate("foo")` would result in several lookups: `foo`, `foo.html`, `foo.txt`, etc.	list of string	`qute.html,qute.txt,html,txt`
🔒 `quarkus.qute.remove-standalone-lines` Specify whether the parser should remove standalone lines from the output. A standalone line is a line that contains only section tags, parameter declarations and whitespace characters.	boolean	`true`
🔒 `quarkus.qute.content-types` The additional map of suffixes to content types. This map is used when working with template variants. By default, the `java.net.URLConnection#getFileNameMap()` is used to determine the content type of a template file.	`Map<String,String>`	required ❗