Quarkus - Qute Reference Guide

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.

1. Hello World Example

In this example, we'd like to demonstrate the basic workflow when working with Qute templates. Let's start with a simple hello world example. We will always need some **template contents**:

hello.html

```
<html>
  <p>Hello {name}! ①
</html>
```

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

Then, we will need to parse the contents into a **template definition** Java object. A template definition is an instance of `io.quarkus.qute.Template`.

If using Qute "standalone" you'll need to create an instance of `io.quarkus.qute.Engine` first. The `Engine` represents a central point for template management with dedicated configuration. Let's use the convenient builder:

```
Engine engine = Engine.builder().addDefaults().build();
```

In Quarkus, there is a preconfigured `Engine` available for injection - see Quarkus Integration.

If we have an `Engine` instance we could parse the template contents:

```
Template helloTemplate = engine.parse(helloHtmlContent);
```

In Quarkus, you can simply inject the template definition. The template is automatically parsed and cached - see Quarkus Integration.

Finally, we will create a **template instance**, set the data and render the output:

```
// Renders <html><p>Hello Jim!</p></html>
helloTemplate.data("name", "Jim").render(); ①
```

① `Template.data(String, Object)` is a convenient method that creates a template instance and sets the data in one step.

So the workflow is simple:

1. Create template contents (`hello.html`),
2. Parse template definition (`io.quarkus.qute.Template`),
3. Create template instance (`io.quarkus.qute.TemplateInstance`),
4. Render output.

The `Engine` is able to cache the definitions so that it's not necessary to parse the contents again and again.

> ℹ️ In Quarkus, the caching is done automatically.

2. Core Features

2.1. Syntax and Building Blocks

The dynamic parts of a template include:

- **Comment**
  - `{! This is a comment !}`,
  - Could be multi-line,
  - May contain expressions and sections: `{! {#if true} !}`.
- **Expression**
  - Outputs the evaluated value,
  - Simple properties: `{foo}`, `{item.name}`,
  - Virtual methods: `{item.get(name)}`, `{name ?: 'John'}`,
  - With namespace: `{inject:colors}`.
- **Section**
  - May contain expressions and sections: `{#if foo}{foo.name}{/if}`,
  - The name in the closing tag is optional: `{#if active}ACTIVE!{/}`,
  - Can be empty: `{#myTag image=true /}`,
  - May declare nested section blocks: `{#if item.valid} Valid. {#else} Invalid. {/if}` and decide which block to render.

2.2. Identifiers

Expressions/tags must start with a curly bracket (`{`) followed by a valid identifier. A valid identifier is a digit, an alphabet character, underscore (`_`), or a section command (`#`). Expressions/tags starting with an invalid identifier are ignored. A closing curly bracket (`}`) is ignored if not inside an expression/tag.

hello.html

```
<html>
    <body>
    {_foo}        ①
    {  foo}       ②
    {{foo}}       ③
    {"foo":true}  ④
    </body>
</html>
```

① Evaluated: expression starts with underscore.

② Ignored: expression starts with whitespace.

③ Ignored: expression starts with `{`.

④ Ignored: expression starts with `"`.

> 💡 It is also possible to use escape sequences `\{` and `\}` to insert delimiters in the text. In fact, an escape sequence is usually only needed for the start delimiter, ie. `\{foo}` will be rendered as `{foo}` (no evaluation will happen).

2.2.1. Expressions

An expression consists of:

- an optional namespace followed by a colon (`:`),

- one or more parts separated by dot (`.`).

The first part of the expression is always resolved against the current context object. If no result is found for the first part it's resolved against the parent context object (if available). For an expression that starts with a namespace the current context object is found using all the available `NamespaceResolver`s. For an expression that does not start with a namespace the current context object is **derived from the position** of the tag. All other parts are resolved using `ValueResolver`s against the result of the previous resolution.

For example, expression `{name}` has no namespace and single part - `name`. The "name" will be resolved using all available value resolvers against the current context object. However, the expression `{global:colors}` has the namespace `global` and single part - `colors`. First, all available `NamespaceResolver`s will be used to find the current context object. And afterwards value resolvers will be used to resolve "colors" against the context object found.

```
{name}  ①
{item.name}  ②
{global:colors}  ③
```

① no namespace, one part - `name`

② no namespace, two parts - `item`, `name`

③ namespace `global`, one part - `colors`

An expression part could be a "virtual method" in which case the name can be followed by a list of comma-separated parameters in parentheses:

```
{item.getLabels(1)}  ①
{name or 'John'}  ②
```

① no namespace, two parts - `item`, `getLabels(1)`, the second part is a virtual method with name `getLabels` and params `1`

② infix notation, translated to `name.or('John')`; no namespace, two parts - `name`, `or('John')`

2.2.1.1. Current Context

If an expression does not specify a namespace the current context object is derived from the position of the tag. By default, the current context object represents the data passed to the template instance. However, sections may change the current context object. A typical example is the for/each loop - during iteration the content of the section is rendered with each element as the current context object:

```
{#each items}
  {count}. {it.name}  ①
{/each}

{! Another form of iteration... !}
{#for item in items}
  {count}. {item.name}  ②
{/for}
```

① `it` is an implicit alias. `name` is resolved against the current iteration element.

② Loop with an explicit alias `item`.

Data passed to the template instance are always accessible using the `data` namespace. This could be useful to access data for which the key is overriden:

```html
<html>
{item.name}  ①
<ul>
{#for item in item.getDerivedItems()}  ②
  <li>
  {item.name}  ③
  is derived from
  {data:item.name}  ④
  </li>
{/for}
</ul>
</html>
```

① `item` is passed to the template instance as a data object.

② Iterate over the list of derived items.

③ `item` is an alias for the iterated element.

④ Use the `data` namespace to access the `item` data object.

2.2.1.2. Built-in Operators

Operator	Description	Examples
Elvis	Outputs the default value if the previous part cannot be resolved or resolves to `null`.	`{person.name ?: 'John'}`, `{person.name or 'John'}`
Ternary	Shorthand for if-then-else statement. Unlike in If Section nested operators are not supported.	`{item.isActive ? item.name : 'Inactive item'}` outputs the value of `item.name` if `item.isActive` resolves to `true`.

In fact, the operators are implemented as "virtual methods" that consume one parameter and can be used with infix notation, i.e. `{person.name or 'John'}` is translated to `{person.name.or('John')}`.

2.2.1.3. Character Escapes

For HTML and XML templates the `'`, `"`, `<`, `>`, `&` characters are escaped by default. If you need to render the unescaped value:

1. Use the `raw` or `safe` properties implemented as extension methods of the `java.lang.Object`,

2. Wrap the `String` value in a `io.quarkus.qute.RawString`.

```
<html>
<h1>{title}</h1> ①
{paragraph.raw} ②
</html>
```

① `title` that resolves to `Expressions & Escapes` will be rendered as `Expressions & Escapes`

② `paragraph` that resolves to `<p>My text!</p>` will be rendered as `<p>My text!</p>`

2.2.2. Sections

A section:

- has a start tag
  - starts with `#`, followed by the name of the section such as `{#if}` and `{#each}`,
- may be empty
  - tag ends with `/`, ie. `{#emptySection /}`
- may contain other expression, sections, etc.
  - the end tag starts with `/` and contains the name of the section (optional): `{#if foo}Foo!{/if}` or `{#if foo}Foo!{/}`,

The start tag can also define parameters. The parameters have optional names. A section may contain several content **blocks**. The "main" block is always present. Additional/nested blocks also start with `#` and can have parameters too - `{#else if item.isActive}`. A section helper that defines the logic of a section can "execute" any of the blocks and evaluate the parameters.

```
{#if item.name is 'sword'}
  It's a sword!
{#else if item.name is 'shield'}
  It's a shield!
{#else}
  Item is neither a sword nor a shield.
{/if}
```

2.2.2.1. Loop Section

The loop section makes it possible to iterate over an instance of `Iterable`, `Map` 's entry set, `Stream` and an Integer. It has two flavors. The first one is using the `each` name alias.

```
{#each items}
   {it.name} ①
{/each}
```

① `it` is an implicit alias. `name` is resolved against the current iteration element.

The other form is using the `for` name alias and can specify the alias used to reference the iteration element:

```
{#for item in items}
   {item.name}
{/for}
```

It's also possible to access the iteration metadata inside the loop:

```
{#each items}
   {count}. {it.name}  ①
{/each}
```

① `count` represents one-based index. Metadata also include zero-based `index`, `hasNext`, `odd`, `even`.

The `for` statement also works with integers, starting from 1. In the example below, considering that `total = 3`:

```
{#for i in total}
   {i}:
{/for}
```

The output will be:

```
1:2:3:
```

2.2.2.2. If Section

A basic control flow section. The simplest possible version accepts a single parameter and renders the content if it's evaluated to `true` (or `Boolean.TRUE`).

```
{#if item.active}
   This item is active.
{/if}
```

You can also use the following operators:

Operator	Aliases	Precedence (higher wins)
logical complement	!	4

Operator	Aliases	Precedence (higher wins)
greater than	`gt`, `>`	3
greater than or equal to	`ge`, `>=`	3
less than	`lt`, `<`	3
less than or equal to	`le`, `<=`	3
equals	`eq`, `==`, `is`	2
not equals	`ne`, `!=`	2
logical AND (short-circuiting)	`&&`, `and`	1
logical OR (short-circuiting)	`||`, `or`	1

A simple operator example

```
{#if item.age > 10}
  This item is very old.
{/if}
```

Multiple conditions are also supported.

Multiple conditions example

```
{#if item.age > 10 && item.price > 500}
  This item is very old and expensive.
{/if}
```

Precedence rules can be overridden by parentheses.

Parentheses example

```
{#if (item.age > 10 || item.price > 500) && user.loggedIn}
  User must be logged in and item age must be > 10 or price must be
> 500.
{/if}
```

You can add any number of `else` blocks:

```
{#if item.age > 10}
  This item is very old.
{#else if item.age > 5}
  This item is quite old.
{#else if item.age > 2}
  This item is old.
{#else}
  This item is not old at all!
{/if}
```

2.2.2.3. With Section

This section can be used to set the current context object. This could be useful to simplify the template structure:

```
{#with item.parent}
  <h1>{name}</h1>   ①
  <p>{description}</p> ②
{/with}
```

① The name will be resolved against the item.parent.

② The description will be also resolved against the item.parent.

It's also possible to specify an alias for the context object:

```
{#with item.parent as myParent} ①
  <h1>{myParent.name}</h1>
{/with}
```

① myParent is the alias that can be used inside the tag.

This section might also come in handy when we'd like to avoid multiple expensive invocations:

```
{#with item.callExpensiveLogicToGetTheValue(true) as it}
  {#if it is "fun"} ①
    <h1>Yay!</h1>
  {#else}
    <h1>{it} is not fun at all!</h1>
  {/if}
{/with}
```

① it is the result of item.callExpensiveLogicToGetTheValue(1,'foo',bazinga). The method is only invoked once even though the result may be used in multiple expressions.

2.2.2.4. Include/Insert Sections

These sections can be used to include another template and possibly override some parts of the template (template inheritance).

Template "base"

```html
<html>
<head>
<meta charset="UTF-8">
<title>{#insert title}Default Title{/}</title> ①
</head>
<body>
  {#insert body}No body!{/} ②
</body>
</html>
```

① `insert` sections are used to specify parts that could be overriden by a template that includes the given template.

② An `insert` section may define the default content that is rendered if not overriden.

Template "detail"

```
{#include base} ①
  {#title}My Title{/title} ②
  {#body}
    <div>
      My body.
    </div>
  {/body}
{/include}
```

① `include` section is used to specify the extended template.

② Nested blocks are used to specify the parts that should be overriden.

> **ℹ** Section blocks can also define an optional end tag - `{/title}`.

2.2.2.5. User-defined Tags

User-defined tags can be used to include a template and optionally pass some parameters. Let's suppose we have a template called `item.html`:

```
{#if showImage} ①
  {it.image} ②
{/if}
```

① `showImage` is a named parameter.

② `it` is a special key that is replaced with the first unnamed param of the tag.

Now if we register this template under the name `item` and if we add a `UserTagSectionHelper` to the engine:

```
Engine engine = Engine.builder()
                .addSectionHelper(new UserTagSectionHelper
.Factory("item"))
                .build();
```

> ℹ️ In Quarkus, all files from the `src/main/resources/templates/tags` are registered and monitored automatically.

We can include the tag like this:

```
<ul>
{#each items}
  <li>
  {#item this showImage=true /} ①
  </li>
{/each}
</ul>
```

① `this` is resolved to an iteration element and can be referenced using the `it` key in the tag template.

2.3. Engine Configuration

2.3.1. Template Locator

Manual registration is sometimes handy but it's also possible to register a template locator using `EngineBuilder.addLocator(Function<String, Optional<Reader>>)`. This locator is used whenever the `Engine.getTemplate()` method is called and the engine has no template for a given id stored in the cache.

> ℹ️ In Quarkus, all templates from the `src/main/resources/templates` are located automatically.

3. Quarkus Integration

If you want to use Qute in your Quarkus application add the following dependency to your project:

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

In Quarkus, a preconfigured engine instance is provided and available for injection - a bean with scope `@Singleton`, bean type `io.quarkus.qute.Engine` and qualifier `@Default` is registered automatically. Moreover, all templates located in the `src/main/resources/templates` directory are validated and can be easily injected.

```java
import io.quarkus.qute.Engine;
import io.quarkus.qute.Template;
import io.quarkus.qute.api.ResourcePath;

class MyBean {

    @Inject
    Template items; ①

    @ResourcePath("detail/items2_v1.html") ②
    Template items2;

    @Inject
    Engine engine; ③
}
```

① If there is no `ResourcePath` qualifier provided, the field name is used to locate the template. In this particular case, the container will attempt to locate a template with path `src/main/resources/templates/items.html`.

② The `ResourcePath` qualifier instructs the container to inject a template from a path relative from `src/main/resources/templates`. In this case, the full path is `src/main/resources/templates/detail/items2_v1.html`.

③ Inject the configured `Engine` instance.

3.1. Injecting Beans Directly In Templates

A CDI bean annotated with `@Named` can be referenced in any template through the `inject` namespace:

```
{inject:foo.price} ①
```

① First, a bean with name `foo` is found and then used as the base object.

All expressions using the `inject` namespace are validated during build. For the expression `inject:foo.price` the implementation class of the injected bean must either have the `price`

property (e.g. a `getPrice()` method) or a matching [template extension method](#) must exist.

> ℹ️ A `ValueResolver` is also generated for all beans annotated with `@Named` so that it's possible to access its properties without reflection.

3.2. Parameter Declarations

It is possible to specify optional parameter declarations in a template. Quarkus attempts to validate all expressions that reference such parameters. If an invalid/incorrect expression is found the build fails.

```
{@org.acme.Foo foo}  ①
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Qute Hello</title>
</head>
<body>
  <h1>{title}</h1>  ②
  Hello {foo.message}!  ③
</body>
</html>
```

① Parameter declaration - maps `foo` to `org.acme.Foo`.

② Not validated - not matching a param declaration.

③ This expression is validated. `org.acme.Foo` must have a property `message` or a matching template extension method must exist.

> ℹ️ A value resolver is also generated for all types used in parameter declarations so that it's possible to access its properties without reflection.

3.2.1. Overriding Parameter Declarations

```
{@org.acme.Foo foo}
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Qute Hello</title>
</head>
<body>
  <h1>{foo.message}</h1>  ①
  {#for foo in baz.foos}
    <p>Hello {foo.message}!</p>  ②
  {/for}
</body>
</html>
```

① Validated against `org.acme.Foo`.

② Not validated - `foo` is overridden in the loop section.

3.3. Template Extension Methods

Extension methods can be used to extend the data classes with new functionality. For example, it is possible to add "computed properties" and "virtual methods". A value resolver is automatically generated for a method annotated with `@TemplateExtension`. If declared on a class a value resolver is generated for every non-private method declared on the class. Methods that do not meet the following requirements are ignored.

A template extension method:

- must be static,

- must not return `void`,

- must accept at least one parameter.

The class of the first parameter is always used to match the base object. The method name is used to match the property name by default. However, it is possible to specify the matching name with `TemplateExtension#matchName()`.

> A special constant - `ANY` - may be used to specify that the extension method matches any name. In that case, the method must declare at least two parameters and the second parameter must be a string.

Extension Method Example

```java
package org.acme;

class Item {

    public final BigDecimal price;

    public Item(BigDecimal price) {
        this.price = price;
    }
}

@TemplateExtension
class MyExtensions {

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

① This method matches an expression with base object of the type `Item.class` and the `discountedPrice` property name.

This template extension method makes it possible to render the following template:

```
{item.discountedPrice} ①
```

① `item` is resolved to an instance of `org.acme.Item`.

3.3.1. Method Parameters

An extension method may accept multiple parameters. The first parameter is always used to pass the base object, ie. `org.acme.Item` in the previous example. Other parameters are resolved when rendering the template and passed to the extension method.

Multiple Parameters Example

```java
@TemplateExtension
class MyExtensions {

    static BigDecimal scale(BigDecimal val, int scale, RoundingMode mode) { ①
        return val.setScale(scale, mode);
    }
}
```

① This method matches an expression with base object of the type `BigDecimal.class`, with the

`scale` virtual method name and two virtual method parameters.

```
{item.discountedPrice.scale(2,mode)}  ①
```

① `item.discountedPrice` is resolved to an instance of `BigDecimal`.

3.3.2. Built-in Template Extension methods

Quarkus provides a set of built-in extension methods.

3.3.2.1. Map extension methods:

- `keys` or `keySet`: Returns a Set view of the keys contained in a map
  - `{#for key in map.keySet}`
- `values`: Returns a Collection view of the values contained in a map
  - `{#for value in map.values}`
- `size`: Returns the number of key-value mappings in a map
  - `{map.size}`
- `isEmpty`: Returns true if a map contains no key-value mappings
  - `{#if map.isEmpty}`
- `get(key)`: Returns the value to which the specified key is mapped
  - `{map.get('foo')}` or `{map['foo']}`

3.3.2.2. Collection extension methods:

- `get(index)`: Returns the element at the specified position in a list
  - `{list.get(0)}` or `{list[0]}`

3.3.2.3. Number extension methods:

- `mod`: Modulo operation
  - `{#if counter.mod(5) == 0}`

3.4. @TemplateData

A value resolver is automatically generated for a type annotated with `@TemplateData`. This allows Quarkus to avoid using reflection to access the data at runtime.

> ℹ️ Non-public members, constructors, static initializers, static, synthetic and void methods are always ignored.

```
package org.acme;

@TemplateData
class Item {

    public final BigDecimal price;

    public Item(BigDecimal price) {
        this.price = price;
    }

    public BigDecimal getDiscountedPrice() {
        return price.multiply(new BigDecimal("0.9"));
    }
}
```

Any instance of `Item` can be used directly in the template:

```
{#each items} ①
  {it.price} / {it.discountedPrice}
{/each}
```

① `items` is resolved to a list of `org.acme.Item` instances.

Furthermore, `@TemplateData.properties()` and `@TemplateData.ignore()` can be used to fine-tune the generated resolver. Finally, it is also possible to specify the "target" of the annotation - this could be useful for third-party classes not controlled by the application:

```
@TemplateData(target = BigDecimal.class)
@TemplateData
class Item {

    public final BigDecimal price;

    public Item(BigDecimal price) {
        this.price = price;
    }
}
```

```
{#each items} ①
  {it.price.setScale(2, rounding)} ①
{/each}
```

① The generated value resolver knows how to invoke the `BigDecimal.setScale()` method.

3.5. RESTEasy Integration

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

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

This extension registers a special `ContainerResponseFilter` implementation so that a resource method can return a `TemplateInstance` and the filter takes care of all necessary steps. A simple JAX-RS resource may look like this:

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.

3.5.1. Variant Templates

Sometimes it could be useful to render a specific variant of the template based on the content negotiation. `VariantTemplate` is a perfect match for this use case:

```java
@Path("/detail")
class DetailResource {

    @Inject
    VariantTemplate item;  ①

    @GET
    @Produces({ MediaType.TEXT_HTML, MediaType.TEXT_PLAIN })
    public Rendering item() {
        return item.data(new Item("Alpha", 1000));  ②
    }
}
```

① Inject a variant template with base path derived from the injected field - `src/main/resources/templates/item`.

② The resulting output depends on the `Accept` header received from the client. For `text/plain` the `src/main/resources/templates/item.txt` template is used. For `text/html` the `META-INF/resources/templates/item.html` template is used.

3.6. Development Mode

In the development mode, all files located in `src/main/resources/templates` are watched for changes and modifications are immediately visible.

3.7. 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`

4. Extension Points

TODO