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.

> In Quarkus, the caching is done automatically.

2. Core Features

2.1. Syntax and Building Blocks

The dynamic parts of a template include:

- **Comment**
  - Starts with `{!` and ends with `!}`: `{! 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.

- **Unparsed Character Data**
  - Starts with `{[` and ends with `]}`: `{[ <script>if(true){alert('Qute is cute!')};</script> ]}`,
  - Could be multi-line,
  - Used to mark the content that should be rendered but not parsed.

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).

2.3. Removing Standalone Lines From the Template

By default, Qute parser removes standalone lines from the template output. A **standalone line** is a line that contains at least one section tag (e.g. `{#each}` and `{/each}`) or parameter declaration (e.g. `{@org.acme.Foo foo}`) but no expression and non-whitespace character. In other words, a line that contains no section tag or a parameter declaration is **not** a standalone line. Likewise, a line that contains an *expression* or a *non-whitespace character* is **not** a standalone line.

Template Example

```
<html>
  <body>
     <ul>
     {#for item in items} ①
       <li>{item.name} {#if item.active}{item.price}{/if}</li>  ②
                                  ③
     {/for}                       ④
     </ul>
   <body>
</html>
```

① This is a standalone line and will be removed.

② Not a standalone line - contains an expression and non-whitespace characters

③ Not a standalone line - contains no section tag/parameter declaration

④ This is a standalone line.

Default Output

```
<html>
  <body>
     <ul>
       <li>Foo 100</li>

     </ul>
   <body>
</html>
```

The default behavior can be disabled by setting the property `quarkus.qute.remove-standalone-lines` to `false`. In this case, all whitespace characters from a standalone line will be printed to the output.

Output with `quarkus.qute.remove-standalone-lines=false`

```
<html>
  <body>
     <ul>

       <li>Foo 100</li>


     </ul>
   <body>
</html>
```

2.3.1. 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')`

2.3.1.1. 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 overridden:

```
<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.

2.3.1.2. 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'}`

Operator	Description	Examples
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`.

The condition in a ternary operator evaluates to `true` if the value is not considered `falsy` as described in the If Section.

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')}`.

2.3.1.3. 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 &amp; Escapes`

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

2.3.1.4. Virtual Methods

A virtual method is a **part of an expression** that looks like a regular Java method invocation. It's called "virtual" because it does not have to match the actual method of a Java class. In fact, like normal properties a virtual method is also handled by a value resolver. The only difference is that for virtual methods a value resolver consumes parameters that are also expressions.

Virtual Method Example

```
<html>
<h1>{item.buildName(item.name,5)}</h1> ①
</html>
```

① `buildName(item.name,5)` represents a virtual method with name `buildName` and two parameters: `item.name` and `5` . The virtual method could be evaluated by a value resolver generated for the following Java class:

```
class Item {
   String buildName(String name, int age) {
      return name + ":" + age;
   }
}
```

Virtual methods are usually evaluated by value resolvers generated for @TemplateExtension methods, @TemplateData or classes used in parameter declarations. However, a custom value resolver that is not backed by any Java class/method can be registered as well.

A virtual method with single parameter can be called using the infix notation:

Infix Notation Example

```
<html>
<p>{item.price or 5}</p>  ①
</html>
```

① `item.price or 5` is translated to `item.price.or(5)`.

Virtual method parameters can be "nested" virtual method invocations.

Nested Virtual Method Example

```
<html>
<p>{item.subtractPrice(item.calculateDiscount(10))}</p>  ①
</html>
```

① `item.calculateDiscount(10)` is evaluated first and then passed as an argument to `item.subtractPrice()`.

2.3.2. 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}
```

2.3.2.1. 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:
```

2.3.2.2. If Section

The `if` section represents a basic control flow section. The simplest possible version accepts a single parameter and renders the content if the condition is evaluated to `true`. A condition without an operator evaluates to `true` if the value is not considered `falsy`, i.e. if the value is not `null`, `false`, an empty collection, an empty map, an empty array, an empty string/char sequence or a number equal to zero.

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

You can also use the following operators in a condition:

Operator	Aliases	Precedence (higher wins)
logical complement	`!`	4
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 also 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}
```

2.3.2.3. 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`.

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

```
{#with item.callExpensiveLogicToGetTheValue(1,'foo',bazinga)}
  {#if this is "fun"} ①
    <h1>Yay!</h1>
  {#else}
    <h1>{this} is not fun at all!</h1>
  {/if}
{/with}
```

① `this` 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.

2.3.2.4. Let/Set Section

This section allows you to define named local variables:

```
{#let myParent=order.item.parent}
  <h1>{myParent.name}</h1>
{/let}
```

The section tag is also registered under the `set` alias:

```
{#set myParent=item.parent price=item.price}
  <h1>{myParent.name}</h1>
  <p>Price: {price}
{/set}
```

2.3.2.5. 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>
<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 overridden by a template that includes the given template.

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

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 overridden.

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

2.3.2.6. 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 `itemDetail.html`:

```
{#if showImage} ①
  {it.image} ②
  {nested-content} ③
{/if}
```

① `showImage` is a named parameter.

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

③ (optional) `nested-content` is a special key that will be replaced by the content of the tag.

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

```
Engine engine = Engine.builder()
                   .addSectionHelper(new
UserTagSectionHelper.Factory("itemDetail","itemDetail.html"))
                   .build();
```

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

We can include the tag like this:

```
<ul>
{#for item in items}
  <li>
  {#itemDetail item showImage=true} ①
    = <b>{item.name}</b> ②
  {/itemDetail}
  </li>
{/for}
</ul>
```

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

② Tag content injected using the `nested-content` key in the tag template.

2.4. Engine Configuration

2.4.1. 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.

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:

```
<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 `@ApplicationScoped`, 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.

```
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. Template Variants

Sometimes it's useful to render a specific variant of the template based on the content negotiation. This can be done by setting a special attribute via `TemplateInstance.setAttribute()`:

```
class MyService {

    @Inject
    Template items; ①

    @Inject
    ItemManager manager;

    String renderItems() {
       return
items.data("items",manager.findItems()).setAttribute(TemplateInstan
ce.SELECTED_VARIANT, new
Variant(Locale.getDefault(),"text/html","UTF-8")).render();
    }
}
```

When using `quarkus-resteasy-qute` the content negotiation is performed automatically. See RESTEasy Integration.

3.2. 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.

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

3.3. 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.

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

3.3.1. 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.

3.4. Template Extension Methods

Extension methods can be used to extend the data classes with new functionality or resolve expressions for a specific namespace. 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 a class is annotated with `@TemplateExtension` then a value resolver is generated for every method declared on the class. Methods that do not meet the following requirements are ignored.

A template extension method:

- must not be `private`
- must be static,
- must not return `void`,
- must accept at least one parameter, unless the namespace is specified.

The class of the first parameter is used to match the base object unless the namespace is specified. In such case, the namespace is used to match an expression. 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 - TemplateExtension#ANY - may be used to specify that the extension method matches any name. In that case, a method parameter is used to pass the property name. If a namespace is specified the method must declare at least one parameter and the first parameter must be a string. If no namespace is specified the method must declare at least two parameters and the second parameter must be a string.

Extension Method Example

```
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`.

3.4.1. Method Parameters

An extension method may accept multiple parameters. If no namespace is specified the first parameter is always used to pass the base object, i.e. `org.acme.Item` in the first example. Other parameters are resolved when rendering the template and passed to the extension method.

Multiple Parameters Example

```
@TemplateExtension
class BigDecimalExtensions {

    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`.

3.4.2. Namespace Extention Methods

If `TemplateExtension#namespace()` is specified then the extension method is used to resolve expressions with the given namespace. Template extension methods that share the same namespace are grouped in one resolver ordered by `TemplateExtension#priority()`. The first matching extension method is used to resolve an expression.

Namespace Extension Method Example

```
@TemplateExtension(namespace = "str")
public static class StringExtensions {

   static String format(String fmt, Object... args) {
      return String.format(fmt, args);
   }

   static String reverse(String val) {
      return new StringBuilder(val).reverse().toString();
   }
}
```

These extension methods can be used as follows.

```
{str:format('%s %s!','Hello', 'world')} ①
{str:reverse('hello')} ②
```

① The output is `Hello world!`

② The output is `olleh`

3.4.3. Built-in Template Extension

Quarkus provides a set of built-in extension methods.

3.4.3.1. 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']}`

3.4.3.2. Collection extension methods:

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

3.4.3.3. Number extension methods:

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

3.5. @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.

> 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.

3.6. 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:

```
<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

```
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.

The content negotiation is performed automatically. The resulting output depends on the `Accept` header received from the client.

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

    @Inject
    Template item; ①

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

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

② 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.

3.7. Development Mode

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

3.8. Type-safe Message Bundles

3.8.1. Basic Concepts

The basic idea is that every message is potentially a very simple template. In order to prevent type errors a message is defined as an annotated method of a **message bundle interface**. Quarkus generates the **message bundle implementation** at build time. Subsequently, the bundles can be used at runtime:

1. Directly in your code via `io.quarkus.qute.i18n.MessageBundles#get()`; e.g. `MessageBundles.get(AppMessages.class).hello_name("Lucie")`
2. Injected in your beans via `@Inject`; e.g. `@Inject AppMessages`
3. Referenced in the templates via the message bundle name; e.g. `{msg:hello_name('Lucie')}`

Message Bundle Interface Example

```
import io.quarkus.qute.i18n.Message;
import io.quarkus.qute.i18n.MessageBundle;

@MessageBundle ①
public interface AppMessages {

    @Message("Hello {name}!") ②
    String hello_name(String name); ③
}
```

① Denotes a message bundle interface. The bundle name is defaulted to `msg` and is used as a namespace in templates expressions, e.g. `{msg:hello_name}`.

② Each method must be annotated with `@Message`. The value is a qute template.

③ The method parameters can be used in the template.

3.8.2. Bundle Name and Message Keys

Keys are used directly in templates. The bundle name is used as a namespace in template expressions. The `@MessageBundle` can be used to define the default strategy used to generate message keys from method names. However, the `@Message` can override this strategy and even define a custom key. By default, the annotated element's name is used as-is. Other possibilities are:

1. De-camel-cased and hyphenated; e.g. `helloName()` → `hello-name`
2. De-camel-cased and parts separated by underscores; e.g. `helloName()` → `hello_name`.

3.8.3. Validation

- All message bundle templates are validated:
  - All expressions without a namespace must map to a parameter; e.g. `Hello {foo}` → the method must have a param of name `foo`
  - All expressions are validated against the types of the parameters; e.g. `Hello {foo.bar}` where the parameter `foo` is of type `org.acme.Foo` → `org.acme.Foo` must have a property of name `bar`

    A warning message is logged for each *unused* parameter.

- Expressions that reference a message bundle method, such as `{msg:hello(item.name)}`, are validated too.

3.8.4. Localization

The default locale of the Java Virtual Machine used to **build the application** is used for the `@MessageBundle` interface by default. However, the `io.quarkus.qute.i18n.MessageBundle#locale()` can be used to specify a custom locale.

Additionaly, there are two ways to define a localized bundle:

1. Create an interface that extends the default interface that is annotated with `@Localized`
2. Create an UTF-8 encoded file located in `src/main/resources/messages`; e.g. `msg_de.properties`.

A localized interface is the preferred solution mainly due to the possibility of easy refactoring.

Localized Interface Example

```
import io.quarkus.qute.i18n.Localized;
import io.quarkus.qute.i18n.Message;

@Localized("de") ①
public interface GermanAppMessages {

    @Override
    @Message("Hallo {name}!") ②
    String hello_name(String name);
}
```

① The value is the locale tag string (IETF).

② The value is the localized template.

Message bundle files must be encoded in UTF-8. The file name consists of the relevant bundle name (e.g. `msg`) and underscore followed by the locate tag (IETF). The file format is very simple: each line represents either a key/value pair with the equals sign used as a separator or a comment (line starts with `#`). Keys are mapped to method names from the corresponding message bundle interface. Values represent the templates normally defined by `io.quarkus.qute.i18n.Message#value()`. We use `.properties` suffix in our example because most IDEs and text editors support syntax highlighting of `.properties` files. But in fact, the suffix could be anything.

Localized File Example - `msg_de.properties`

```
hello_name=Hallo {name}! ① ②
```

① Each line in a localized file represents a message template.

② Keys and values are separated by the equals sign.

3.9. 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 ❗