C++ Template Precompiler

Einführung

Dies ist ein in Java Standard Edition (<https://www.oracle.com>) geschriebenes Tool, um auf HTML, CSS und Javascript basierende Templatedateien in C++-Code zu kompilieren. Dieser Code ist abhängig von folgenden Technologien/Libraries:

- C++17

- FastCGI <https://github.com/toshic/libfcgi.git>

- Qt 5

- aktivweb WebAppCommon Library

- aktivweb QtCommon2 Library

Installation

1. Voraussetzung ist eine vorhandene Java Runtime Environment 1.8
2. Zum Komprimieren bzw. Optimieren von Javascript bzw. CSS wird auf Drittprogramme zurückgegriffen, beide basieren ebenfalls auf Java:
   1. Javascript: **Google ClosureCompiler**: <https://github.com/google/closure-compiler>
   2. CSS: **YUI Compressor**: <https://github.com/yui/yuicompressor>   
      Es wird empfohlen, die Version 2.4.7 statt 2.4.8 aufgrund von Bugs in letztgenannter Version zu verwenden
3. Die Pfade zu den jeweiligen .jar-Dateien der Drittprogramme sind in der Datei **settings.dat** im Installationsverzeichnis einzutragen, relative Pfade sind möglich

Ordnerstruktur im Webapp-Qt-Projekt

1. **css**: einzubindende CSS-Dateien (für Inline-Rendering)
2. **js**: einzubindende Javascript-Dateien (für Inline-Rendering)
3. **layout\_templates**: Übergeordnete Layout-Templates, etwa für Menü-Strukturen
4. **repository**:Downloadverzeichnis für Remote JS/CSS bzw. Speicherort komprimierte JS/CSS-Dateien
5. **subtemplates**: HTML-Dateien, die innerhalb Templates eingebunden werden können (vgl. z.B. Partials bei TYPO3-Fluid)
6. **templates**: die HTML-Dateien enthalten den Teil der HTML-Ausgabe, die innerhalb des <body>-Tags gerendert werden soll. Als Wrapper können Layout-Templates eingebunden werden

XML-Konfigurationsdateien

Je Projekt muss eine XML-Konfigurationsdatei angelegt werden, wo die Pfade und die zu kompilierenden Dateien registriert werden. Ein Beispiel:

<?xml version="1.0" encoding="UTF-8"?>

<templateconfig src="template" dest="view">

<template name="FirstPage" />

<template name="SecondPage" />

<template path="Ajax" name="Dialog" />

<template path="Ajax" name="Dialog2" renderToString="true" />

</templateconfig>

* <templateconfig>
  1. **src**: Basis-Quellverzeichnis
     + der Pfad kann relativ zum Verzeichnis sein, wo die XML-Konfigurationsdatei sein (wird empfohlen)
     + in dem Verzeichnis sind Unterverzeichnisse
  2. **dest**: Basis-Zielverzeichnis
     + heißt üblicherweise nach dem MVC-Muster „view“
     + sämtliche kompilierten Klassen werden in das Unterverzeichnis c**ompiledtemplate** geschrieben
* <template>
  1. **name**: entspricht dem Namen der Templatedatei. Z.B.: wenn die Templatedatei im src-Verzeichnis **FirstPage.html** heißt, so ist hier FirstPage anzugeben
  2. **path**: Ein oder mehrere Unterverzeichnisse, in dem die Templatedatei liegt
  3. **renderToString**: Es erfolgt keine direkte Ausgabe auf dem FastCGI Output Stream, sondern es wird ein QString zurückgegeben. Das ist nützlich zum Aufbauen von JSON-Ausgaben für AJAX-Aufrufe

Kompilieren der Templates

- Der Aufruf lautet wie folgt:

java -jar cpptemplate.jar [Optionen] <Pfad zur XML-Konfigurationsdatei>

* Optionen
  + --debug  
    Entwicklungsmodus: Kein Komprimieren von Javascript und CSS
  + --nocache  
    Standardmäßig werden CSS und JS-Dateien nicht neu komprimiert, wenn sie nicht geändert wurden bzw. Remote-Dateien (über HTTP) nicht neu heruntergeladen. Die Angabe von --nocache verhindert dies und komprimiert die Dateien jedes Mal
* Beispiel:   
  java -jar cpptemplate.jar --debug --nocache D:\Projekte\WebApp\template\_config.xml  
    
  (mit Templates im Verzeichnis D:\Projekte\WebApp\template)

Templateelemente

- es sind nach dem HTML-Standard gültige Tags und Attribute möglich. Darüber hinaus gibt es diese Konstrukte:

1. <cpp:include>  
   Wenn dieses Tag angegeben wird, dann ist es als erstes Element im Template anzugeben
   * <cpp:baseTemplate>: Einbindung von Layout-Templates. Details siehe unten  
     Attribute
     + name: Name bzw. Pfad zur Layout-Template-Datei ohne Dateiendung .html
   * <cpp:js>  
     Bindet ein Javascript entweder als Inline-Element oder als URL ein. Download von externen URLs als inline-Script und relative Pfade zum Basisverzeichnis sind möglich
     + Attribute

* **src**: Pfad/URL
* **includeType**: *inline* oder *link*. Wenn Attribut nicht angegeben, dann per Default *inline*
  + - Beispiele:  
      <cpp:js src="https://code.jquery.com/jquery-3.3.1.min.js" includeType ="link"/>  
      <cpp:js src="js/script.js"/>
  + <cpp:css>  
    Inline CSS (style-Tag), analog wie a.
    - Attribute
* **src**: Pfad/URL  
  + - Beispiel:  
      <cpp:css src="css/bootstrap.min.css" includeType ="link"/>  
      <cpp:css src="css/my-style.css"/>
  + <cpp:header file="*pfad/zur/headerdatei.h*"/>  
    Einbindung von C++-Headern, etwa für statische Aufrufe von Methoden

1. <cpp:renderSubtemplate
   * Bindet html-Dateien an dieser Stelle ein, die im Ordner **subtemplates** liegen
   * Attribute
     + name: Pfad relativ zu Ordner **subtemplates** ohne Dateiendung .html
2. Ausgabe von Strings mit automatischem HTML Escaping mit geschweiften Klammern  
   - Der Ausdruck muss den Typ QString haben  
   {*ausdruck*}: wird intern gewandelt zu FastCgiOutput::write(*ausdruck*.toHtmlEscaped())
3. Direkte Ausgabe ohne Escaping von Ausdrücken mit doppelten geschweiften Klammern  
   - Der Ausdruck kann diese Datentypen haben: QString, const char\*, bool, int, double, float  
   {{*ausdruck*}}: wird intern gewandelt zu FastCgiOutput::write(*ausdruck*)  
     
   Abweichend von der C++-Syntax sind an dieser Stelle inline-if-Anweisungen ohne then-Teil möglich, d.h. das inline-if wird in einen Standard-if-Block umgewandelt. Anwendungsfall z.B. Ausgabe einer zusätzlichen CSS-Klasse bei einer Bedingung, und keine Ausgabe im Else-Fall:  
   <div class='styleclass{itemActive ? " active"}'>  
   ...  
   </div>
4. <cpp:if>,<cpp:then>,<cpp:elseif>,<cpp:else>  
   - Konditionale Ausdrücke: Ausgabe erfolgt nur, wenn die Bedingung zutrifft. Falls es nur den then-Fall gibt, ist das Auslassen des <cpp:then>-Tags zulässig  
   Attribute
   * cond (bei cpp:if und cpp:elseif): beliebiger boolescher Ausdruck  
     Beispiele:  
     Einfaches if  
     <cpp:if cond="data->isLoggedIn()">  
      <button>Ausloggen</button>  
     </cpp:if>

If-then und Else

<cpp:if cond="data->hasError()">

<cpp:then>  
 Es ist ein Fehler aufgetreten

</cpp:then>

<cpp:else>  
 Erfolgreich gespeichert  
</cpp:else>  
</cpp:if>

If-then und else if, aber ohne Else-Fall

<cpp:if cond="data->getCase() == 1">

<cpp:then>  
 Fall 1

</cpp:then>

<cpp:elseif cond="data->getCase() == 2">  
 Fall 2

</cpp:elseif>  
</cpp:if>

1. <cpp:for>
   * Definition von Schleifen (foreach und Zählschleifen): Wiederholte Ausgabe von weiteren Template-Elementen
   * Attribute
     + Falls foreach-Schleife
       - **each**: Angabe der zu durchlaufenden Datenstruktur
       - **as**: Variablenname des aktuellen Elements
     + Falls Zählschleife
       - **def**: Angabe der Definition der Schleife und Abbruchbedingung in C++-Syntax
   * Beispiele
     + Foreach  
       <ul>  
       <cpp:for each="data->getItems()" as="item">  
        <li>{item->getName()}</li>  
       </cpp:for>  
       </ul>
     + Zählschleife  
       <ul>  
       <cpp:for def="int i=0;i<data->getItemCount();i++">  
        <li>{{i+1}}. {data->getItem(i)->getName()}</li>  
       </cpp:for>  
       </ul>
2. Direktes Einbinden von C++-Code mit Spezial-Tags (in Analogie zur <?php-Syntax)  
     
   <?cpp // hier C++-Code ?>
3. Kommentare (werden komplett entfernt aus präkompilierter cpp-Datei)
   * HTML-Syntax: <!-- -->
   * Über Tags <cpp:comment><cpp:comment>

Einbindung der erzeugten Klassen in C++-Code

* das komplette HTML-Dokument wird von der vordefinierten Klasse HtmlTemplate (aus Library WebAppCommon, Header: mvc/view/html/htmltemplate.h), der Inhalt des Body-Tags wird durch die in C++-Code umgewandelten Templates, Subtemplates und Layout-Templates erzeugt
  + Beispiele
    - Rendern des HTML-Codes in der update-Methode einer View-Klasse, die von einer aus den HTML-Templates automatisch generierten Klasse erbt:  
        
      void DemoPageView::update(unique\_ptr<ViewData> viewdata)   
      {   
       /\* Hilfsmethode zum Downcasten von Unique-Pointern, definiert in Library  
       QtCommon2, Header: core/cast.h  
       \*/  
       unique\_ptr<DemoPageViewData> data=cast<DemoPageViewData,ViewData>(viewdata);  
         
        
       // render wurde aus den Templates erzeugt  
       DemoPageCompiledTemplate::render(std::move(data), out);  
        
        
       // renderFooter ist in der Basisklasse HtmlTemplate definiert

this->renderFooter();  
}

* Für Ajax-Aufrufe wird üblicherweise nur der Templateinhalt gerendert (d.h. ohne DOCTYPE und HTML-Header), im Fall von Ajax-Aufrufe mit JSON-Rückgabewert kann das Template zu einem QString gerendert werden und dann als JSON-Wert eingefügt werden