Tntnet users guide

Authors: Tommi Mäkitalo, Andreas Welchlin

Table of contents

Concept

Tntnet is a application server for web applications written in c++. A web application in tntnet is written with a template-language, which embeds c++-processing-instructions in html-pages. You can use exernal classes or libraries in these pages. That way programmers can concentrate on the html-result, when creating content and put processing in c++-classes.

Applications written with the template-language called ecpp are compiled into c++-classes and linked into a dynamically loaded shared library. This is done at compile-time – not at runtime, like other template-systems often do. On runtime there is no compiler needed.

Installing Tntnet

Tntnet runs on Linux/Unix. You need a c++-compiler to compile Tntnet and also to compile your web applications. As a prerequisite cxxtools (which is available through the Tntnet-homepage http://www.tntnet.org) is needed. Cxxtools is a collection of useful c++-classes.

To install tntnet you have to:

1. install cxxtools
2. unpack the sources with “tar xzf tntnet-version.tar.gz”
3. cd to the source-directory “cd tntnet-version.tar.gz”
4. run “./configure”
5. run “make”
6. run “make install”

This installs: The application server and tools for

1. tntnet – the web application server
2. ecppc – the ecpp-compiler
3. ecppl – the ecpp-language-extractor for internationalization
4. ecppll – the ecpp-language-linker for internationalization
5. some shared libraries
6. tntnet-config – a script, which gives you information about the installed Tntnet and helps you setting up a simple project.

Create a simple application with Tntnet

The simples way to create a tntnet-application is to use the tntnet-config-script. Just run “tntnet-config –project=projectname”. This creates a directory with the name of the project and:

1. projectname.ecpp – your first ecpp-file
2. Makefile – a simple makefile to build the application
3. tntnet.conf – a basic configuration file for tntnet
4. tntnet.properties – a basic configuration file for logging-configuration

You should use a valid c++-class name as the project name, because the project name is used as a

class name.

The script tells you some basic instructions how to build and run the project.

To build the project, change to the directory and run “make”. It should compile out of the box. You get a shared library projectname.so, which contains the web application. To run it start “tntnet -c tntnet.conf” and navigate your browser to http://localhost:8000/projectname. You should see a simple Web-page.

C++-content (processing, expressions, conditional expressions)

There are some simple tags, with which you can embed c++-content into the page. The most important are processing-tags and output-tags.

With processing-tags you insert c++-code into the page, which is processed on runtime. There are 3 options to insert code:

The most verbose version is <*%cpp*>...some c++-code...</*%cpp*>. A newline after </*%cpp*> is ignored.

<*{* starts an inline-block, which is terminated by *}*>. A newline after the closing tag is not ignored, but passed to the browser.

The character '*%*' in the first line starts also c++-code until the end of line.

To prevent interpretation of these you can always precede the tag with '\', which prevents the interpretation of the next character.

You can embed a c++-expression with <*$ expr $*> into the html-code. The result of the expression is printed into the page. The type of the expression needs to have a output-operator for std::ostream. The output is not regarded as html. Characters with special meaning in html are translated into their html-entities.

Often you need to print something depending on a condition. You can put a if-statement around it. As a shortcut there is a special tag <*? cond ? output ?*>. “cond” is evaluated as a c++-expression. If the result of this expression is true, *output* is printed. The output is translated like <*$...$*>.

Another useful tag is <*# ... #*>, which is just a comment. The content is skipped by the ecpp-compiler.

In C++-mode *reply.out()* returns a *std::ostream*, which writes text to the html-page. *reply.sout()* returns a *std::ostream*, which escapes characters with special meaning in html before writing to the page.

Query-arguments (scalar/vector, untyped/typed, default-value)

Web-applications need to interact with the user. Therefore they send a html-form to the user. After the user submits the form, the application must interpret the content of the form.

To support this, ecpp has a special tag: <*%args*>...</*%args*>. Between this pair you define the arguments of the form. Each argument is terminated with ';'. Ecpp generates c++-variables of type std::string, containing the content of the form.

You can precede arguments with a c++-type to convert the parameter automatically into this type.

This is done using the input-operator of std::istream.

Ecpp supports multiple input-tags with the same name, e.g. multiple checkboxes or select-tag with attribute *multiple*. By appending [] to your argument a std::vector and a typedef will be generated. Writing *name[]* will be compiled to a typedef *name_type* and a vector *name*.

Single arguments can have a default value using the syntax *variable = default_value;*.

Examples:

```
<%args>
name;
street;
city = "New York";
int age;      // content of text-input is converted to int
int sport[];  // multiple checkboxes with the same name on my form
</%args>
<# use the vector like this: #>
% for (sport_type::const_iterator it = sport.begin(); it !=
sport.end(); ++it) {
<$ *it $>
% }
```

Components

Ecpp-pages are called **components**. They are identified by their name. The name is composed of the class-name and the library-name divided with '@'.

Components, which are called by Tntnet are called top-level-components.

Top-level-components return the http-response-code. If a explicit return-statement is not specified the constant HTTP_OK is returned. Constants, which define http-response-codes are defined in the header tnt/http.h.

Components can contain internal **subcomponents**. The subcomponent-name is appended to the class-name divided by a dot.

Examples are:

`mycomp@app`
identifies a component with the name "mycomp" in the shared library "app.so"

`mycomp.subcomp@app`
identifies a subcomponent "subcomp" of "mycomp@app"

Component-parameters

Every component has 3 parameters called "request", "reply" and "qparam".

The "request"-parameter contains information about the request, received from the client. This includes information about http-headers, the peer-ip, cookies or multipart-components. It is defined as "tnt::httpRequest".

The "reply"-parameter is used to build the answer to the request. The http-answer can be modified here. The reply-object is defined as "tnt::httpReply". It contains methods to manipulate the http-headers and an output-stream for writing to the http-body.

"qparam" specifies the query-parameters of the component. For top-level-components it contains the query-parameters of the form. As described above, the parameters are accessed via the <%args>-

block.

Calling components (dynamic/static)

Normally you don't want to write a whole web-application in one file, but you want to split it into pieces and glue them together, just like you would write normal applications. You don't write a single function, but split it into smaller parts and call them from a “main”-function.

This paradigm is supported by using components. Components can be called from other components. You can write reusable components, which implement some specific parts of your page, e. g. a table or select-box.

Components are called using <*&* ... >.

Basically there are several possibilities to call a component.

In the simplest case the component-name without library-part can be put inside these tags.

Component-names without library-part are searched in the local subcomponent first. If not found, the component is searched in the library of the calling component.

> Example:
>
> ```
> <& somecomp >
> ```
> call the component “somecomp” here. If there is a internal subcomponent, this is called, otherwise the component is looked up in the library of the calling component.

Subcomponents, which are defined in other components, are called by appending the subcomponent-name to the component-name separated by a dot.

> Example:
>
> ```
> <& othercomp.subcomp >
> ```
> call the subcomponent “subcomp” in the component “othercomponent”.

To call a component in another library the library-part is appended to the name.

> Example:
>
> ```
> <& somecomp@somelib &>
> ```
> inserts the component “somecomp” from “somelib.so” here

Component-names can be computed at runtime. To call a computed component-name, the expression is put inside brackets.

> Example:
>
> ```
> % std::string comp = “comp”;
> <& (comp + “@otherlib”) &>
> ```
> The component “comp@otherlib” is called.

Calling components from c++

Components can be called directly from C++ using the method *callComp*. *callComp* takes the parameters

- component-id (of type *tnt::compident* or *tnt::subcompident* or a *std::string*)
- request

- reply
- query-parameter-object (of type *cxxtools::query_params*)

It returns a http-result-code. See the API-documentation for details.

Sometimes it is useful not to send the output directly to the client. With the method *scallComp()* it is also possible to retrieve the output and e.g. modify it before sending. It takes the same parameters as *callComp*, except *reply*. Instead of that a temporary reply-object is passed. The http-return-code of the called component is ignored.

Example:

```
std::string result = scallComp(request, qparam);
```

Include ecpp-files

Ecpp-files can be included using the tag <%include>filename</%include>. The content is included at compile-time. It is similar to the #include-directive in C++.

Including Ecpp-files can be useful for:

- global declarations
- initialization, which is needed in multiple components
- it is strongly recommended for global-scope (explained below) variables.

It should not be used for including content. Component-calling is better for that, because the content is not duplicated.

Declaring subcomponents

Subcomponents are declared using <*%def compname*>...</*%def*>. The syntax of subcomponents is the same as in top-level-components. The only exception is, that it is not possible to define other subcomponents there.

Example:

```
Hello <& who >

<%def who>
World
</%def>
```

Prints “Hello World”.

Passing parameters to components

Subcomponents receive named query-parameter just like top-level-components. The parameters are passed inside the <&...>-tags. The called component defines the parameters using an <%args>-block.

Example:

```
<& greeting name=”Linus” lastname=”Torvalds”>

<%def greeting>
<%args>
```

```
lastname;
name;
</%args>
Hi <$ name $> <$ lastname $>
</%def>
```

Prints:

```
      Hi Linus Torvalds
```

Expressions are enclosed in brackets. The parameter type can also be numeric or any other type, which is serializable and deserializable using std::ostream and std::istream.

Example:

```
<{
  std::string nameValue = "Linus";
  std::string lastnameValue = "Torvalds";
  unsigned repeatNum = 3;
}>
<& greeting name=(nameValue) lastname=(lastnameValue)
repeat=(repeatNum)>

<%def greeting>
<%args>
lastname;
name;
unsigned repeatNum = 1;
</%args>
% for (unsigned n = 0; n < repeatNum; ++n) {
Hi <$ name $> <$ lastname $>
% }
</%def>
```

Prints:

```
      Hi Linus Torvalds
      Hi Linus Torvalds
      Hi Linus Torvalds
```

A component can pass all its parameters to a subcomponent using its own qparam-object. In the example below the parameters of the subcomponent *greeting* are forwarded to *printname*.

Example:

```
<& greeting name="Linus" lastname="Torvalds">

<%def greeting>
<h1>greeting</h1>
<&printname qparam>
</%def>

<%def printname>
<%args>
lastname;
name;
</%args>
Hi <$ name $> <$ lastname $>
<%def>
```

Prints:

```
<h1>greeting</h1>
Hi Linus Torvalds
```

Defining Variables

Variables are defined in normal c++-syntax by specifying type and name of the variable and the termination character ';'. These variables are instantiated automatically on first use. Constructor-parameters are specified by adding them in brackets after the name.

Lifetime

Variables are defined in a lifetime-area. The lifetime can be session, request, application or thread.

Session-variables are valid for the current user session. Sessions are implemented using cookies.

Example

```
<%session>
std::string currentUser;
</%session>
```

Request-variables are valid while the current request is processed. This is the shortest possible lifetime.

Example

```
<%request>
unsigned nextId(0);
</%request>
```

Application-variables are valid as long as tntnet is running. They are shared between users. Locking of application-variables is done by tntnet. This is explained in detail in chapter “multi threading”.

Example

```
<%application>
DbConnection myConnection(“customerdb”);
</%application>
```

To use the application-lifetime correctly it is necessary to know, that your application is one shared library. Application-variables are only accessible within the same shared library.

Thread-variables are valid, as long as the thread is running. These variables are not shared between threads, but each thread has his own copy.

Example

```
<%thread>
tntdb::Connection myConnection(“sqlite:customer.db”);
</%thread>
```

Scope

State-variables are by default valid only in the component, where they are specified. The same name can be used in different components without conflicts.

Sometimes it us useful to widen this scope. You could e.g. define a session-variable “currentUser”,

which is accessible throughout your application. The scopes are specified by adding the scope-attribute to the tag. Valid scopes are: component (the default), page (inside the current component and its internal subcomponents) and global.

The application is a shared library with your components.

Examples:

```
<%session scope="global">
std::string currentUser;
</%session>
<%request scope="page">
int number(1);
</%request>
<%def>
<%request scope="page">
int number;  // this references the variable "number" defined
             // outside this subcomponent
</%request>
</%def>
```

Cookies

Cookies are supported by tntnet with a simple api. A cookie consists of a name, a value and optionally attributes. To set a cookie a ecpp-application call "reply.setCookie". The simplest form is to call it with 2 parameters: a name and a cookie. You can pass just a std::string or a character-string as name and value of the cookie. When the browser does its next request, you can read the cookie with "reply.getCookie()". This returns a cookie-object, which is directly convertible to a std::string.

Example:

To set a cookie:

```
<{
  reply.setCookie("mycookie", "myvalue");
}>
```

At next request retrieve the value:

```
<{
  std::string v = request.getCookie("mycookie");
  // "v" has the value "myvalue" now
}>
```

To clear the cookie:

```
<{
  reply.clearCookie("mycookie");
}>
```

Component attributes

Sometimes it is useful to query attributes of a component. Attributes are just string-values, which are defined inside a component.

Attributes are defined within a <%attr>-block. A attribute consists of a name, the character '=' and a string terminated by ';'.

To query attributes of a component you need to fetch a reference to it with fetchComp. Pass a component identifier to fetchComp and use the method getAttribute with a name-parameter to

retrieve the value.

Example:

Define a attribute:

```
<%attr>
myattribute = “myvalue”;
</%attr>
```

To query the attribute:

```
<{
  std::string v =
fetchComp(“component@library”).getAttribute(“myattribute”);
  // “v” contains the value “myvalue” now.
}>
```

Configuration

Tntnet needs a configuration file to run. By default this is read from “/etc/tntnet/tntnet.conf”, but you can pass a different file with the command line-switch -c, as was done at the first example.

The file is a text file and contains configuration variables. Every line starts with a variable name followed by 0 or more parameters separated by whitespace. If a parameter contains itself whitespace, enclose the parameter in single or double quotation marks. If you need the quotation mark in the value, you must precede it with '\' to escape its special meaning.

Lines starting with '#' and empty lines are ignored.

There are variables, which are read by tntnet. Unknown variables are ignored. Components might use them, so they need not be unknown if they are unknown to tntnet.

The most important variable is “MapUrl”. It tells tntnet, what to do with requests. Without this variable tntnet answers every request with 404 – HTTP_NOT_FOUND. “MapUrl” takes at least 2 parameters: a regular expression, which is matched against the url, sent from the client and a component name, which to call, if the expression matches. The component name might contain back references to the regular expression.

Examples:

```
# maps every html-file to the component with the same basename e.g.
# /index.html     index@myapp
MapUrl /(.*).html $1@myapp

# maps requests, which end with .jpg to components with _jpg-suffix
e.g.
# /myimg.jpg => myimg_jpg@myapp
MapUrl /(.*).jpg  $1_jpg@myapp

# makes every component available through http by mapping the part
before
# first '/' to the applicationname (shared-library-name) and the part
after
# the app
MapUrl /(.*)/(.*) $2@$1
```

Accessing configuration variables

Sometimes web applications need some configuration e.g. a database-url. Instead of hard coding it, the application can put it into the configuration file of tntnet.

Configuration variables read this way can contain only a single value.

To specify a variable, put a <%config>-block into your component. Inside define a variable my putting the name and optionally a default value separated with '=' and terminate the definition with ';'.

Example:

```
<%config>
dburl = “database=mydb”;
</%config>
```

To set the dburl-parameter put a line:

```
dburl “database=anotherdb”
```

into tntnet.conf.

If you don't specify a default value, the variable is set to an empty string, when not set in tntnet.conf.

Logging

Logging is done through the cxxtools-meta-logging-library. This uses log4cplus, log4cxx or a simple builtin-logging depending on configuration of cxxtools. You don't have to bother too much about the used library, when developing tntnet-applications. The API stays the same. If you use just the cxxtools-provided macros, you can switch later to another library without changing the source of your application.

Every logging-library cxxtools supports, support logging categories and severities. Tntnet defines for each component a category “component.*componentname*”. When you need to log something, just use one of the macros log_fatal, log_error, log_warn, log_info or log_debug. The parameter is passed to a output stream and you can put multiple outputitems separated by '<<'.

What is logged where is specified in “tntnet.properties” (but this can be changed in tntnet.conf). Just look into the example tntnet.properties for some details about configuration and look at the documentation of the underlying logging-library.

Exception handling

Exceptions derived from std::exception thrown by components are catched by tntnet. They usually generate a 500 – HTTP_INTERNAL_SERVER_ERROR, but there is a exception class tnt::httpError, which takes a error code and a message. The code is used as a http-error-code. You should use the tnt::HTTP_*-constants for it.

Savepoints

Sometimes you might want to catch exceptions generated by lower level classes or subcomponents

and generate a nicely formatted error-message. In this case you might want to close all open html-tags before starting your error-message. This is often a almost impossible task, because you don't know, where the exception occurred. Luckily you get help from tntnet/ecpp to solve this open-html-tag problem: savepoints. A savepoint is a simple class, which acts like a transaction for html-output. You instantiate a savepoint as a local variable at the start of your try-block and pass the request-object to it. Before catch call savepoint::commit(). If the commit is not reached, because a exception happened, the savepoint-object rolls back the output to the point, where it was instantiated.

Example:

```
<{
  try
  {
    tnt::savepoint mysavepoint;
}>
   <form>
    <input type="text" name="value" value="<$ object.getValue() $>">
   </form>
<{
    mysavepoint.commit();
  }
  catch (const your_dbexception& e)
  {
}>
    <p class="error">A database-error occured: <$e.what$></p>
<{
  }
}>
```

In case object.getValue() throws a "your_dbexception" the form-tag is discarded from the output of the component and a nice error-message is printed. Without the savepoint the output ends at "value="". This results in a badly formatted html-page.

Binary content

Web applications may contain logos and other images or binary content. They often have static content. For web applications to be complete, you need a way to include your images.

In Tntnet/ecpp you can generate components out of binary files, e.g. jpeg-images. The ecpp-compiler ecppc has a special switch -b, which generates c++-classes out of these files, without interpreting tags, which might appear in binary files. With the switch -m you can specify the mimetype, but if you don't do that, ecppc will use your mime-database (normally /etc/mime.types).

Example:

To generate a component out of a jpeg-file logo.jpg:

```
ecppc -b logo.jpg
```

will generate a logo.h and logo.cpp, which you can compile an link into your application library.

The downside is, that for small binaries like small graphics there is a relative large overhead (I measured a increase of the web application of about 8k per image). Therefore starting with version 1.5 of tntnet the ecpp-compiler is able to pack multiple binaries into a single component. This is done with the switch -bb.

To address a image in a multi-image-component, you have to pass the name of your image as pathinfo to MapUrl in your tntnet.conf:

```
MapUrl /images/(*.jpg)  mymultiimagecomponent@mywebapp $1
```

Upload

Uploading files is supported by html with the input-tag with type=”file”. To use it, you have to specify the enctype “multipart/form-data” in your form-tag.

When the form is submitted, the values comes in a special format, which supports large objects. This is supported by tntnet and it has a simple api to access the uploaded file.

Because files might be somewhat larger than other form-content, they are not passed through the qparam-parameter and not accessible in <%args>-blocks. You have to fetch a const reference to a tnt::multipart-object with request.getMultipart(). This object is a container of objects of type tnt::part. You can either iterate through the parts or use the find-method to directly find the part, which contains the uploaded file. The tnt::part-object gives you access to the content either through constant iterators or just as a string. The iterator-interface is more efficient, because it does not need to instantiate a temporary string. You should consider always to use the iterator-interface.

Example:

To create a form with a upload-field:

```
<form enctype=”multipart/form-data”>
 <input type=”file” name=”upload”>
 <input type=”submit”>
</form>
```

To save the uploaded file:

```
<{
  const tnt::multipart& mp = request.getMultipart();
  tnt::multipart::const_iterator it = mp.find("upload");
  if (it != mp.end())
  {
    std::ofstream f(“result.dat”);
    std::copy(it->getBodyBegin(), it->getBodyEnd(),
      std::ostreambuf_iterator<char>(f));
  }
}>
```

Using c++-classes

As mentioned earlier it is desirable to separate content and processing. Content is the view, which is created using ecpp. Processing are some c++-classes. What you need is a way to access c++-classes from your ecpp-files. Luckily this is easy done.

A c++-class consists generally of a interface in a header-file and a implementation in a implementation-file. You can link your application with the implementation, but you need a way to include the interface into your ecpp-file. This is done in the <%pre>-block. Contents of this block is copied unmodified to the start of the generated header. You can place #include-directives here.

Example:

```
<%pre>
#include “yourclass.h”  // include the class-definition here
</%pre>
<{
  yourclass c; // e.g. instantiate your class
}>
```

```
<p><$ c.getAttribute1() $></p>  <# include a attribute-value into your
page #>
```

You might as well link your shared library to a external library and use it this way.

Some notes about multi threading

When developing ecpp-components you have to bear in mind, that Tntnet is multithreaded, so your code have to be thread-safe. Luckily Tntnet does it's best to encapsulate most of it, but if you use your own classes this must be considered.

You can use cxxtools::Mutex and cxxtool::MutexLock to protect your objects.

Scoped variables are automatically protected. If you use a application-scoped variable the current request locks the application-scope-object and prevents other concurrent threads accessing the application-scope until the request is finished. Using session-scope locks the application-scope-object first and then session-scope-object to prevent deadlocks, where one request waits for the session-scope and the other wait for the application-scope.