XMLmind XML Editor Web Edition - Manual

Hussein Shafie

XMLmind Software
35, rue Louis Leblanc
78120 Rambouillet
France
Phone: +33 (0)9 52 80 80 37
xmleditor-support@xmlmind.com
www.xmlmind.com/xmleditor/

May 2, 2025

Table of Contents

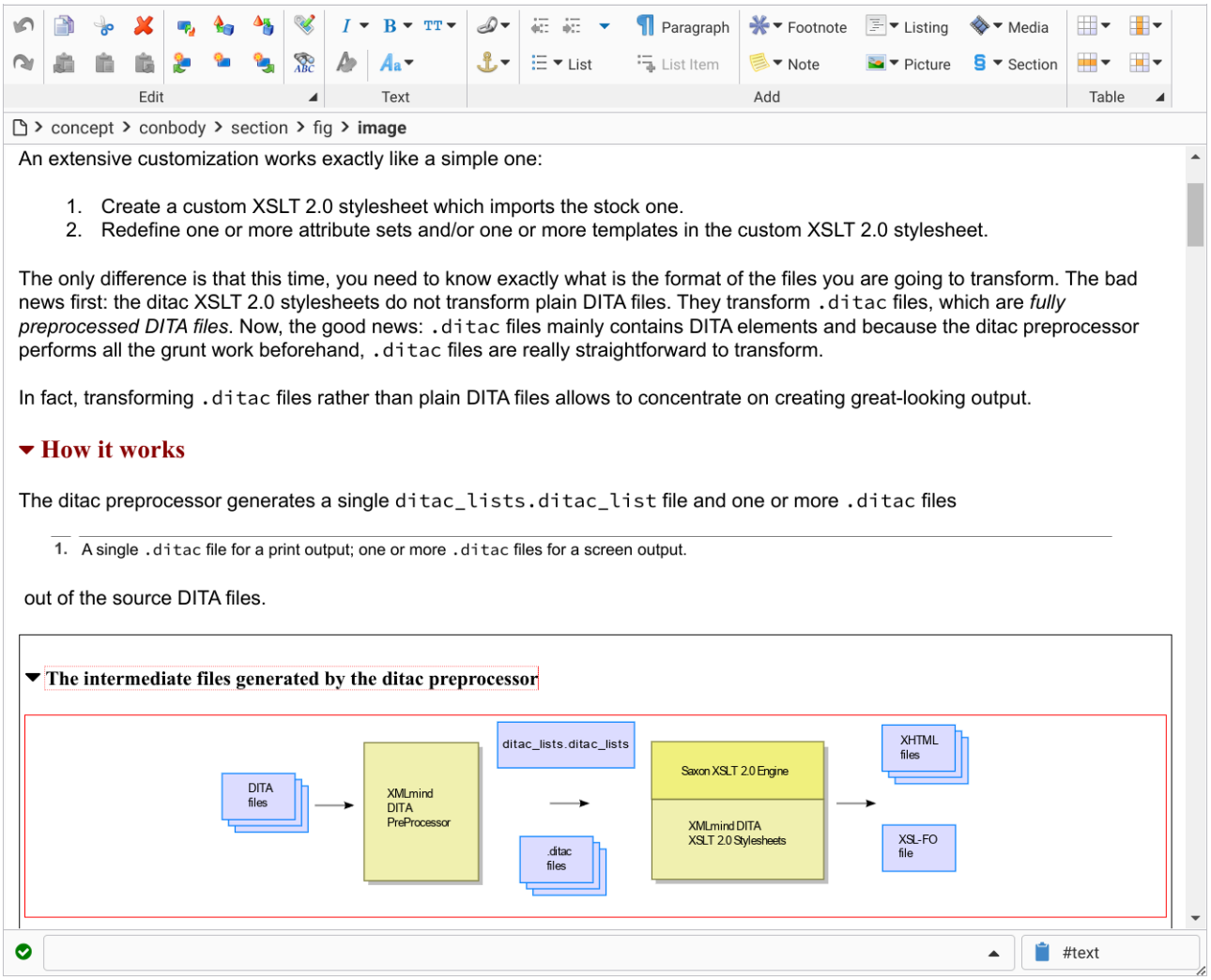
Part I. What is XMLmind XML Editor Web Edition?

What exactly is **XMLmind XML Editor Web Edition**? How does it work? Learn about its strengths and its weaknesses and decide whether it's worth giving this product a try.

Chapter 1. Presentation

XMLmind XML Editor Web Edition (**XXEW** for short) is a JavaScript implementation of XMLmind XML Editor running in the web browser, thus not requiring any installation on the user side.

*Figure 1-1. A DITA `<concept>` opened in **XXEW***



Who will use it?

- **XXEW** is a strictly validating, near WYSIWYG, XML editor, featuring a *streamlined*, *single document*, user interface and having out of the box support for DITA, DocBook, XHTML and TEI Lite.
- **XXEW** is definitely not a programmer's tool and is intended to be used by technical writers, engineers and scholars in order to author *topics* —relatively small, relatively independent from each other, document chunks— which are part of large modular documents.

Who will deploy it?

- **XXEW** is essentially a 100% JavaScript, *lightweight*, software component which has been designed to be easily integrated into any information system (e.g. a **CMS**). As such it aims to serve the same purpose as rich text editors like TinyMCE or CKEditor, but in the context of structured editing. See Part II, Chapter 6, Section 1. Overview.

- An XML editor web application is included in the software distribution. Therefore, out of the box, **XXEW** may also be used to edit local and/or remote XML files. See The sample XML editor application included in the **XXEW** distribution.

Differences with the desktop application

Desktop Application	Web Edition
Requires installing the application on the user's computer. (No need to install Java™. A private Java runtime is included in most software distributions.)	• Requires installing a *very recent* web browser on the user's computer. **Restriction** At the time of this writing only *very recent* Blink-based browsers like Google Chrome or Microsoft Edge and Gecko-based browsers like Firefox are supported. Apple Safari, which uses the WebKit engine, is currently not supported. • Requires installing a Java 11+ runtime and **XXEW** distribution on the server side and running `xxeserver`, which is **XXEW** backend. More about all these requirements in Chapter 2. How it works.
Multiple document user interface, adapted to authoring large, complex, modular, documents, including DITA maps or DocBook assemblies.	Single document user interface, adapted to authoring topics, articles, chapters, sections, etc. **Restriction** Related restrictions: • **XXEW** has less toolbar buttons, less menus, less menu items than its desktop counterpart. It also has less keyboard shortcuts and its keyboard shortcuts are somewhat different. • **XXEW** cannot be used to insert *element references* (e.g.

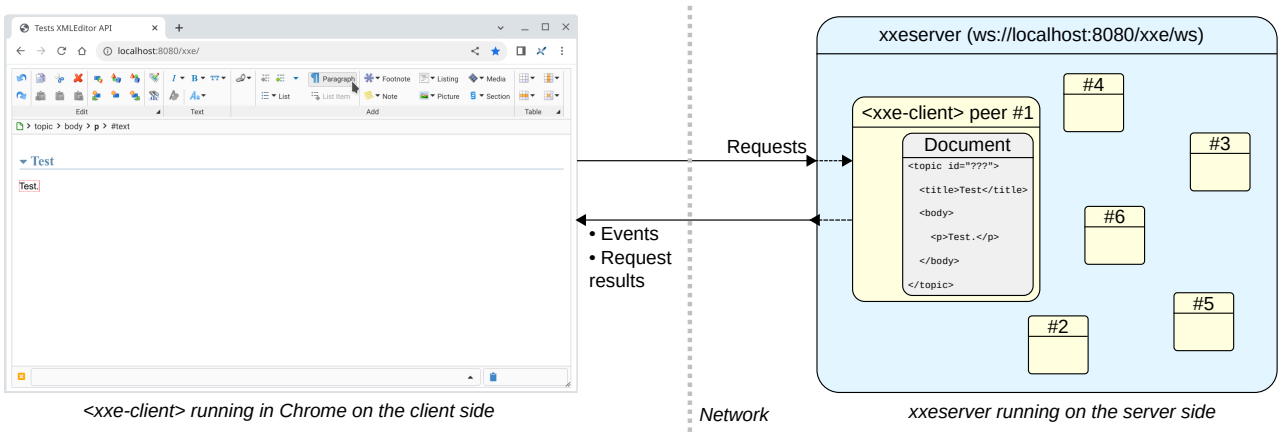
Desktop Application	Web Edition
	XInclude, DITA `@conref`) into a document.
No restrictions related to “local files”.	When editing a document stored in a file which is local to the computer running the web browser, **XXEW** cannot render image references (e.g. DITA `<image href="..."/>`) and cannot transclude element references. The reason is that, for security reasons, a web browser gives a web application very little access to the local file system.
Multiple views of the document being edited may be displayed side by side. These views are the tree view, styled views (each view being specified using a different CSS stylesheet), with or without visible tags, and the XML source view.	Only a single view of the document being edited is displayed at a time. It's possible to switch between the tree view and one of the styled views. Visible tags are not supported. The XML source view is not supported.
Spell checking is `@lang`/`@xml:lang` aware and automatically switches between dictionaries.	Spell checking is implemented by the web browser, which is not convenient to use in the context of multi-lingual documents.
Has advanced import DOCX, paste from MS-Word and paste from web browser facilities. Can convert XML to a variety of formats (PDF, Web Help, EPUB, RTF, ODT, DOCX, etc.)	Has no import or export facilities.
Has CJK (Chinese, Japanese, and Korean) support. Has right-to-left writing (Arabic, Persian, Hebrew, etc) support.	Typing text using a CJK *Input Method Editor* (IME) works but has limitations and bugs[(1)]. No right-to-left writing support.
Is also a MathML WYSIWYG editor.	MathML rendered on screen (by the web browser), but editable only using the tree view.
On Linux, the *X Window Primary Selection* is natively supported.	On Linux, the X Window Primary Selection is not natively supported by **XXEW**. This differs from HTML `<input type="text">` and `<textarea>` and may be surprising for the user of the web browser. However, an optional —crude— emulation is available and works on all platforms.

(1) For example, it's not possible to replace the text selection simply by tying text using the IME.

Chapter 2. How it works

Unlike rich text editors like TinyMCE or CKEditor, XMLmind XML Editor Web Edition (**XXEW** for short) is ***not a standalone*** program entirely written in JavaScript. **XXEW** consists in two programs: `<xxe-client>`, a 100% JavaScript frontend running in the web browser and **`xxeserver`**, a Java™ application backend running on a server computer.

*Figure 2-1. **XXEW** architecture*



`<xxe-client>` cannot function without being connected to an **`xxeserver`** through the WebSocket ("`ws://`" URL) protocol or preferably, the WebSocket Secure ("`wss://`" URL) protocol.

`<xxe-client>` is lightweight and thus loads quickly in the web browser. It does just two things: display as HTML+CSS a view of the XML document being edited and interact with the user.

`xxeserver` does *everything else*: load, validate, modify, save, close the XML document, execute commands received from `<xxe-client>` in order to modify the XML document, compute which HTML +CSS representing the view of the XML nodes is to be sent to `<xxe-client>`, etc. **`xxeserver`** is in fact a server variant of the proven XMLmind XML Editor Desktop Application. Of course, as a full-fledged server program, **`xxeserver`** can run on headless server computers and can handle multiple, concurrent `<xxe-client>`s.

Benefits of this architecture

- `<xxe-client>` is lightweight[2] and thus loads quickly in the web browser.
- Being just a server variant of the XMLmind XML Editor desktop application, **`xxeserver`** shares with the desktop application almost all its code, commands, configurations[3], add-ons and user preferences. This also means that fixing a bug or enhancing the desktop application will almost certainly fix the same bug or improve **`xxeserver`** in the same way.
- Because the state of `<xxe-client>` —including the XML document being edited— is maintained by **`xxeserver`** (see "`<xxe-client>` peer" in the figure above), this state can be fully automatically recovered when needed too[4]. For example, if the user of `<xxe-client>`clicks the "**Go back**"

[2] `<xxe-client>` is a custom HTML element. Its implementation comprises about 7 000 lines of CSS and 17 000 lines of JavaScript (non obfuscated, non minified) at the time of this writing.

[3] More information in "*How to adapt an existing "`.xxe`" configuration file to **XXEW***".

[4] This feature is so useful and so reassuring to the user that it is turned on by default. See boolean attribute `@autorecover` of custom HTML element `<xxe-client>`.

button of the browser and then clicks "**Go forward**", then she/he will automatically find `<xxe-client>` as she/he left it. Same reassuring behavior if the user clicks the "**Reload current page**" button of the browser or if she/he closes and then reopens the browser tab/window containing `<xxe-client>`.

Related information

- Appendix A. How to adapt an existing "`.xxe`" configuration file to **XXEW**

Part II. Deploying XMLmind XML Editor Web Edition

Learn how to deploy **XMLmind XML Editor Web Edition**, whether a 5 minutes demo or a production level deployment. Also learn how to integrate an XML editor into your own web application (for JavaScript programmers).

Chapter 3. Installing XMLmind XML Editor Web Edition

Requirements

On the server side (computer running **`xxeserver`**, the backend of **XXEW**):

- Multi-core computer having at least 8Gb[(5)] of RAM. The largest number of processor cores and the largest amount of memory, the best.
- Officially supported only on: Windows 10+ 64-bit, macOS 14.x (Sonoma) and 15.x (Sequoia) Intel® or Apple® Silicon processor and Linux.
- Java™ 11+.
- Ports 18078 and 18079 (secure connection) which are used by default by **`xxeserver`** to listen to client connections must not be blocked by your firewall.

On the client side (computer running the web browser):

- A *very recent version of Google Chrome* or any browser using the same Blink browser engine: Edge, Opera, Brave, etc. Firefox works fine too, but without system clipboard integration. (Safari is currently not supported. All mobile web browsers are definitely not supported.)
- Ports 18078 and 18079 (secure connection) which are used by default to connect to **`xxeserver`** must not be blocked by your anti-virus, firewall, proxy, etc.

Installing a software distribution

Unpack the **XXEW** distribution inside any directory you want.

- On Windows, unzip the `xxe-web-*-win.zip` distribution. This distribution contains in `bin/jre64/`, a very recent —generally the most recent— *private* OpenJDK Java™ runtime. Therefore no need to install Java on the Windows computer running **`xxeserver`**.
- On the Mac, unpack the `xxe-web-*-mac.tar.gz` distribution. This distribution contains in `bin/jre/` (for Macs having an Intel® processor) and in `bin/jrea/` (for Macs having an Apple® Silicon processor), very recents —generally most recent— *private* OpenJDK Java™ runtimes. Therefore no need to install Java on the Mac running **`xxeserver`**.
- On Linux and other Java™ 11+ platforms, unzip the `xxe-web-*.zip` distribution.

  Make sure that you have a Java™ 11+ runtime installed on your machine. To check this, open a terminal and type "`java -version`" followed by `Enter`.

```
~$ java -version
openjdk version "24" 2025-03-18
OpenJDK Runtime Environment (build 24+36-3646)
OpenJDK 64-Bit Server VM (build 24+36-3646, mixed mode)
```

Contents of the installation directory

The installation directory contains code and resources which are common to XMLmind XML Editor Desktop Edition (**XXE**) and XMLmind XML Editor Web Edition (**XXEW**). The code and resources which are specific to **XXEW** are found in subdirectory `web/`.

(5) By default, **`xxeserver`** is configured to consume at most 2Gb.

addon/
: This `addon/` directory contains a number of add-ons which are bundled with **XXE**.

addon/config/
: Contains configuration files for a number of document types: DocBook, DITA, XHTML, etc.

bin/
: Contains **XXE** code (`.jar` files).

legal/, legal.txt
: Contains legal information about third-party components used in **XXE**.

web/
: Code and resources specific to **XXEW**.

  bin/
  : Contains `xxeserver.jar`, the code of **`xxeserver`** and scripts used to start **`xxeserver`**. Use shell script **`xxeserver`** on the Mac and on Linux. Use **`xxeserver.bat`** and **`xxeservice.exe`** on Windows.

  doc/
  : Contains **XXEW** documentation.

  etc/
  : Empty directory which may be useful when running **`xxeserver`** (could contain a self-signed certificate, a remote file access JSON specification file, etc).

  legal/, legal.txt
  : Contains legal information about **XXEW** and about third-party components used in **XXEW**.

  lib/
  : All the Java™ class libraries needed to run **`xxeserver`**.

  var/
  : Empty directory which may be useful when running **`xxeserver`** (typically contains logs).

  webapp/index.html
  : An HTML page containing the sample XML editor web application. This makes the **XXEW** distributions ready to use out of the box without having to configure or program anything.

  webapp/xxeclient/
  : The CSS and JavaScript™ code of `<xxe-client>` and `<xxe-app>`.

Chapter 4. A quick demo on a single computer

Start `xxeserver`

1. Open a command prompt (Windows) or a terminal (Mac, Linux).
2. Go to directory *`XXE_INSTALL_DIR`*`/web/bin/`, *`XXE_INSTALL_DIR`* being the directory where XMLmind XML Editor Web Edition (**XXEW** for short) has been installed.
3. Run **`xxeserver.bat`** (Windows) or **`xxeserver`** (Mac, Linux shell script).

```
C:\...\web\bin> xxeserver.bat
```

   - **`xxeserver`** should run fine on any platform supporting Java™ 11+.

> **Tip**
>
> The Windows `.zip` distribution and the Mac `.tar.gz` distribution contain a private copy of the most recent version of the Java runtime. Therefore, there is generally no need to install Java on the computer running **`xxeserver`**.

> **Note**
>
> If **`xxeserver`** does not start, please refer to Troubleshooting: **`xxeserver`** does not start.

   - As explained in Part I, Chapter 2. How it works, **`xxeserver`** is mainly a WebSocket server. However it has also been made an HTTP server in order to be able to run the sample XML editor application described below without having to install anything other than **XXEW**.
   - By default, **`xxeserver`** does not support secure connections (`https://`, `wss://` URLs) and listens to HTTP and WebSocket requests on port 18078. Of course, these simple settings can be changed. See Chapter 7. **`xxeserver`** command-line options.
4. At the end of the demo, simply type `Ctrl-C` in the command prompt or terminal to stop **`xxeserver`**.

Open the page containing the sample XML editor application in your browser

1. Start a web browser on the computer running **`xxeserver`**[(6)].

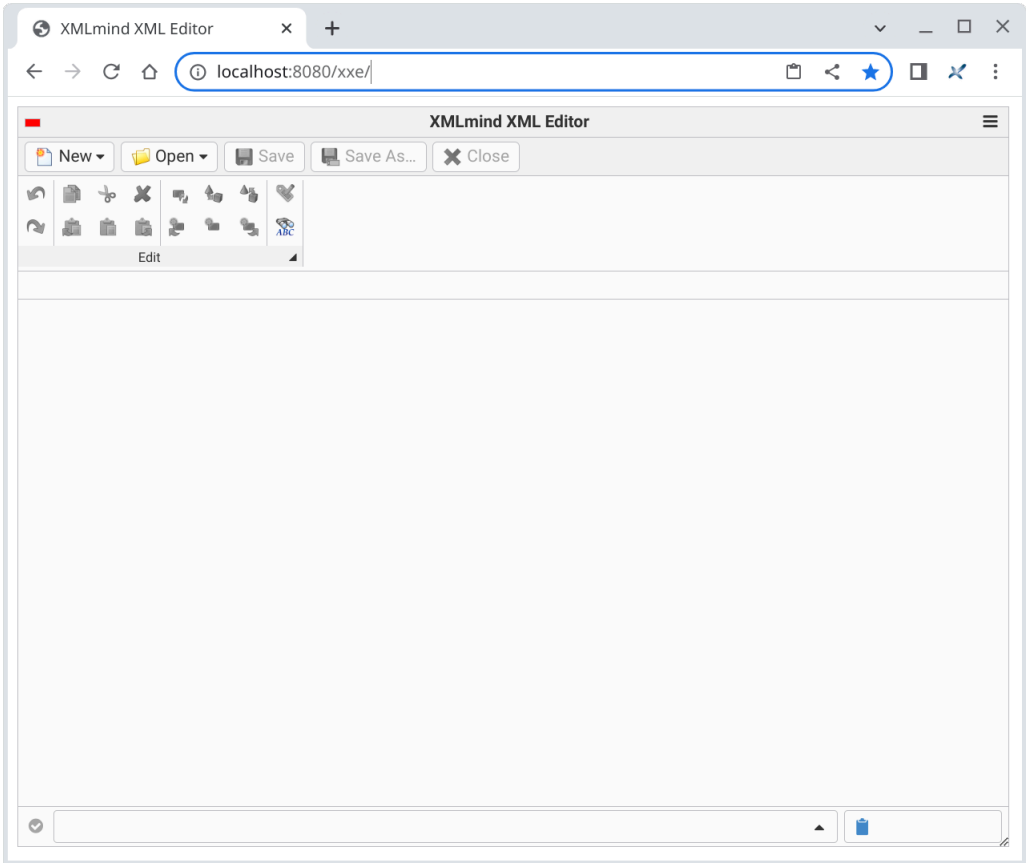
> **Important**
>
> At the time of this writing only *very recent* Blink-based browsers like Google Chrome or Microsoft Edge and Gecko-based browsers like Firefox are supported. Apple Safari, which uses the WebKit engine, is currently not supported.

(6) Please remember that this is nothing more than just a quick, 5 minutes demo. It's by no means a real world use case.

We recommend using Google Chrome or Microsoft Edge because these browsers currently have the best support for editing local files and for integrating the system clipboard with the XML editor.

2. In the address bar of the web browser, please type "`http://localhost:18078/xxe/`".
3. A sample XML editor application based on `<xxe-client>` is now ready to use.

Figure 4-1. A sample XML editor application based on `<xxe-client>`



Note

If the sample XML editor application does not load or does not work, please refer to Troubleshooting: the sample XML editor web application does not work.

The sample XML editor application included in the XXEW distribution

The **XXEW** distribution includes a sample XML editor application. This application lets you create or modify XML documents found:

- On the computer running the web browser. These are called *local files*.
- On the computer running **`xxeserver`**. These are called *remote files*.

In the case of this quick demo, these two computers are the same.

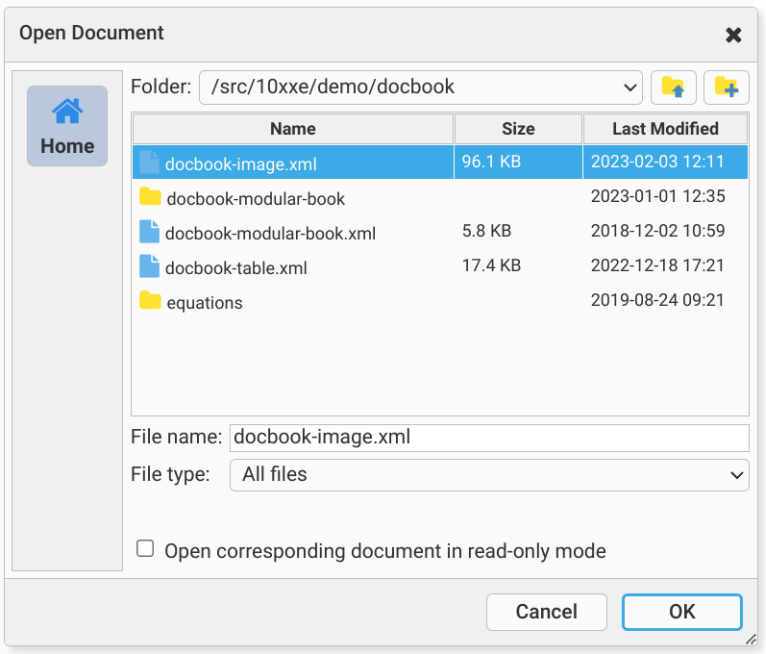
Which remote files may be accessed by **XXEW** and how these files are accessed —read-write or read-only— may be configured. See Chapter 7. **`xxeserver`** command-line options. In the case of this quick

demo, **XXEW** has a read-write access to any file found in the home directory of the user who started **`xxeserver`**.

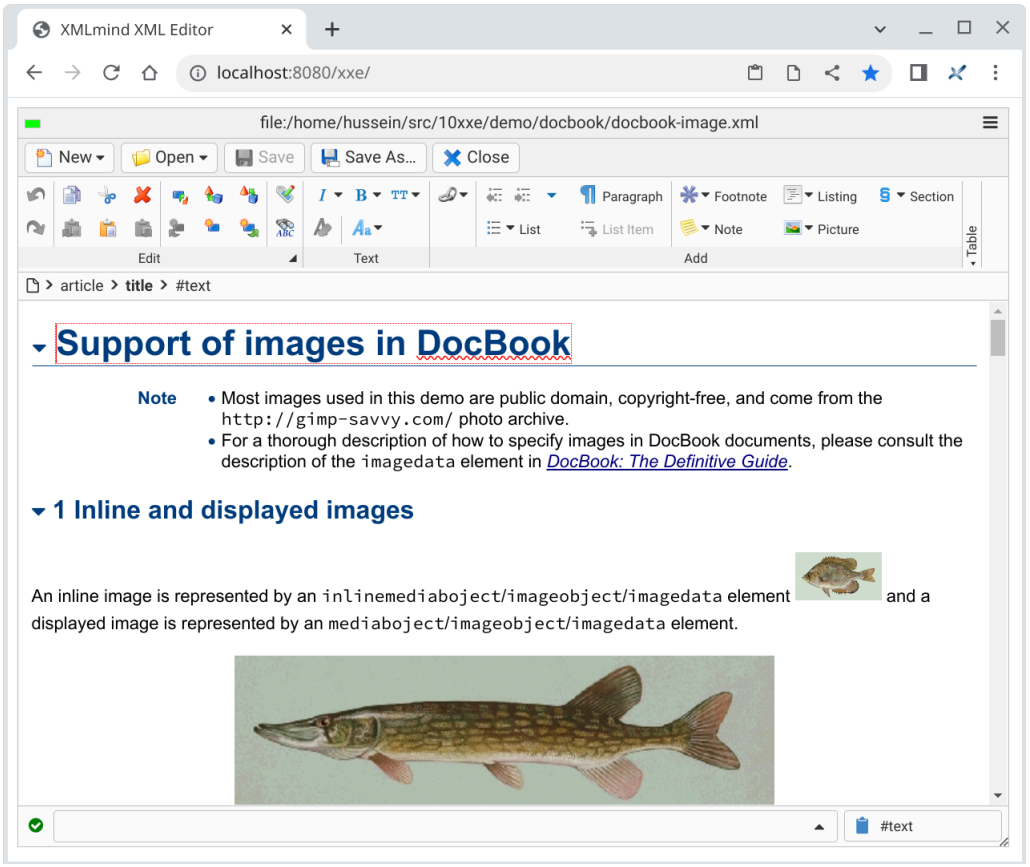
While opening or saving remote files is seamless and works like in any desktop application, the same cannot be said for local files. For security reasons, the browsers give web applications like the sample XML editor very limited access to the local file system. On most browsers, the access to the local file system is even *minimal*. For example, on browsers other than Google Chrome (or Microsoft Edge), **Save** is equivalent to **Save As**.

Opening a DocBook document as a remote file

1. Click **Open** and select "**Open Remote Document**". The Remote File Chooser is displayed.

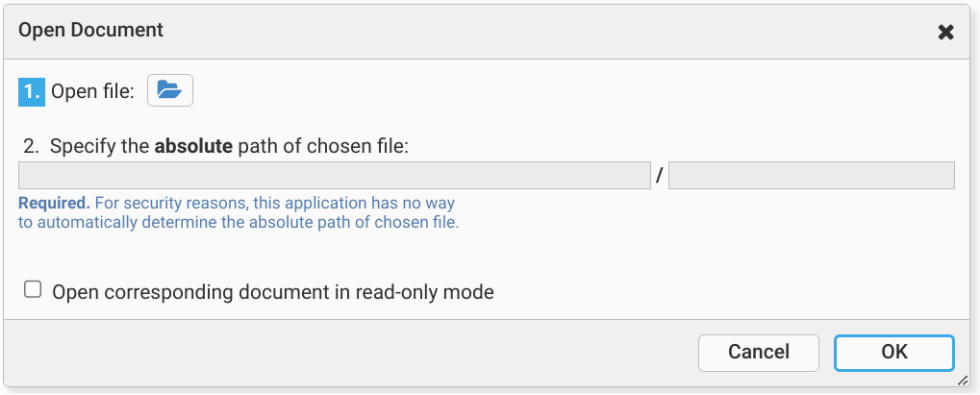


2. Select a remote XML file then click **OK**. The corresponding document is opened in the XML editor.



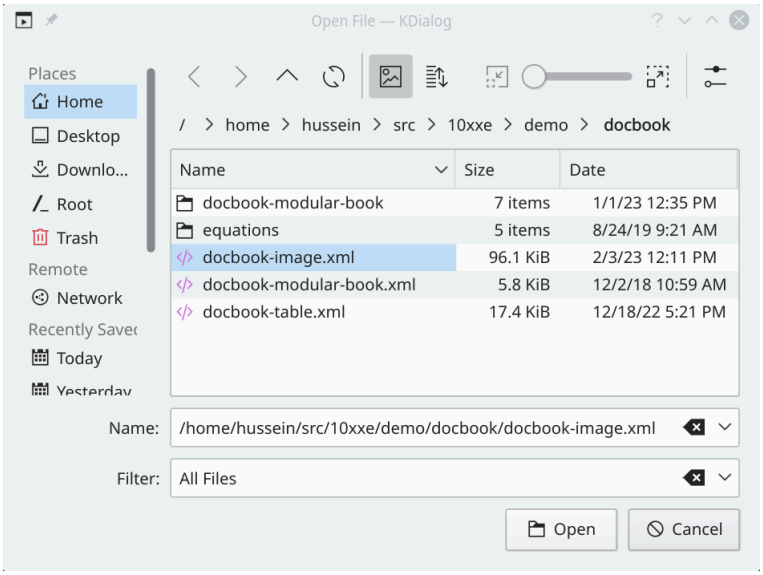
Opening a DocBook document as a local file

1. Click **Open** and select "**Open Local Document**". The Local File Chooser is displayed.

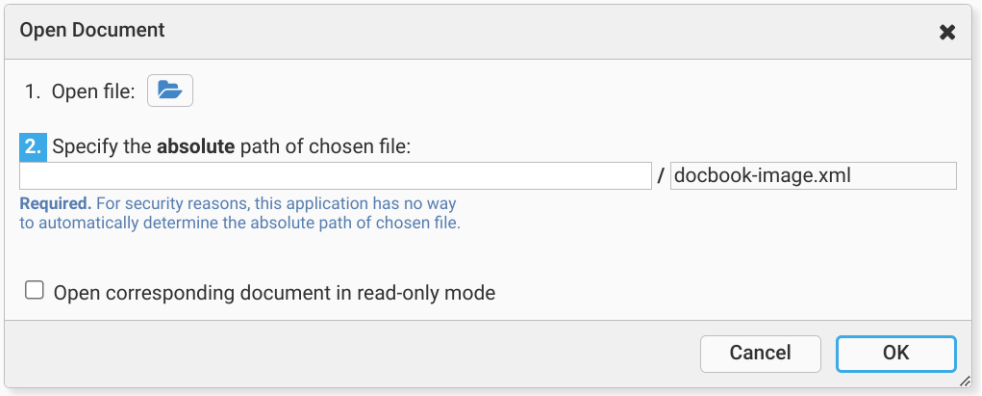


2. Choosing a local file involves ***two*** steps.

   2.a. Click the "**Open file**" button. This displays the "**Open File**" dialog box of the web browser.

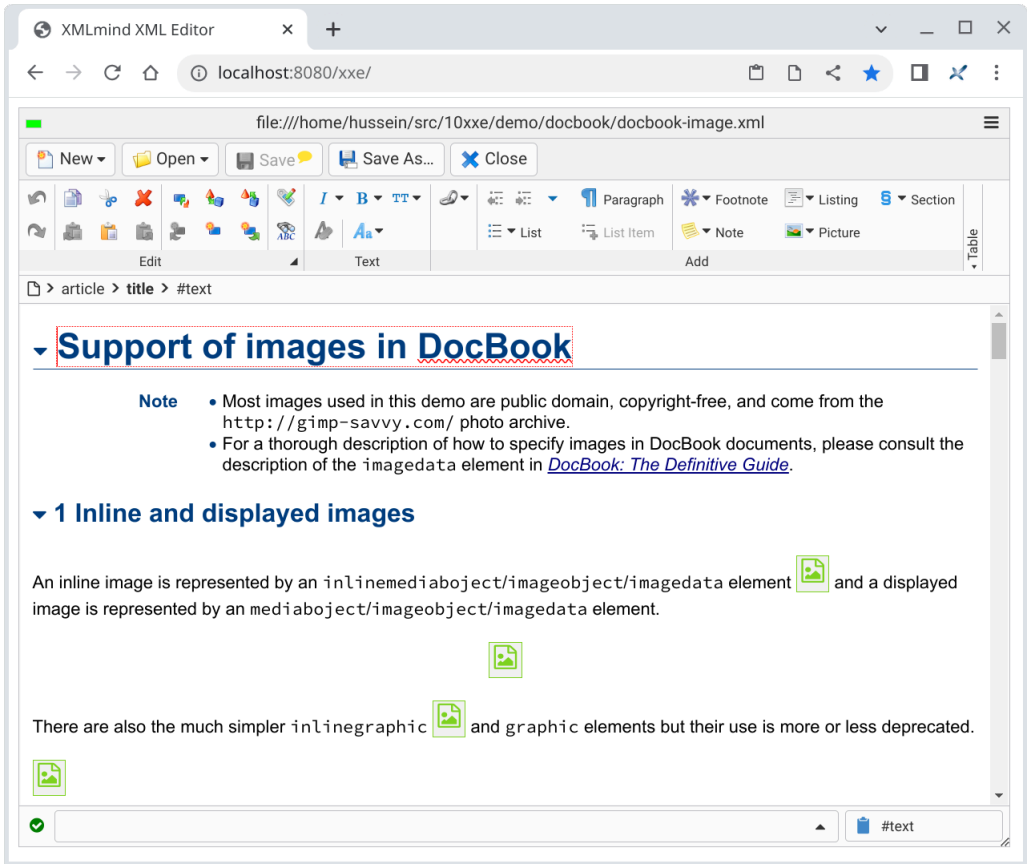


2.b. Select an XML file and also use the facility of this dialog box to copy the path of the directory containing selected file, then click **OK**. The Local File Chooser now suggests to proceed to step #2.



2.c. Type[7](or paste) the file path of the directory containing selected file ("`/home/hussein/src/10xxe/demo/docbook`" in this example). On some web browsers, notably when saving a document, you'll also have to type the name of selected file ("`docbook-image.xml`" in this example). The corresponding document is opened in the XML editor.

[7] The file paths and file names you type in this dialog box are remembered across editing sessions. This means that you can pick file paths and file names from the text field autocompletion lists rather than type the same values over and over.



Notice that all the images found in the document are displayed as green "Picture" icons.

Remember

When you insert an image into a document opened as a local file, you'll be able to see the inserted image. However, because the web browser gives web applications very limited access to the local file system, if you close the document and then reopen it, the newly inserted image is now represented by  , a green *image placeholder* icon. This is normal. Nothing to worry about.

A possible workaround is to embed the image in the document rather than simply reference its file. (**XXEW** lets you do this quite easily.) However, you must keep in mind that embedding images may create huge XML files and also may cause XML interchange problems.

Related information

- Chapter 5. Deploying the sample XML editor

Chapter 5. Deploying the sample XML editor

Why an HTTPS connection is *really* needed

The **XXEW** distribution includes a simple yet useful sample XML editor application. In the previous chapter, you learned how to deploy it on a single computer, that is, **`xxeserver`** and the web browser hosting the sample XML editor both running on the same computer. The URL of the HTML page containing the sample XML editor was: `http://localhost:18078/xxe/`.

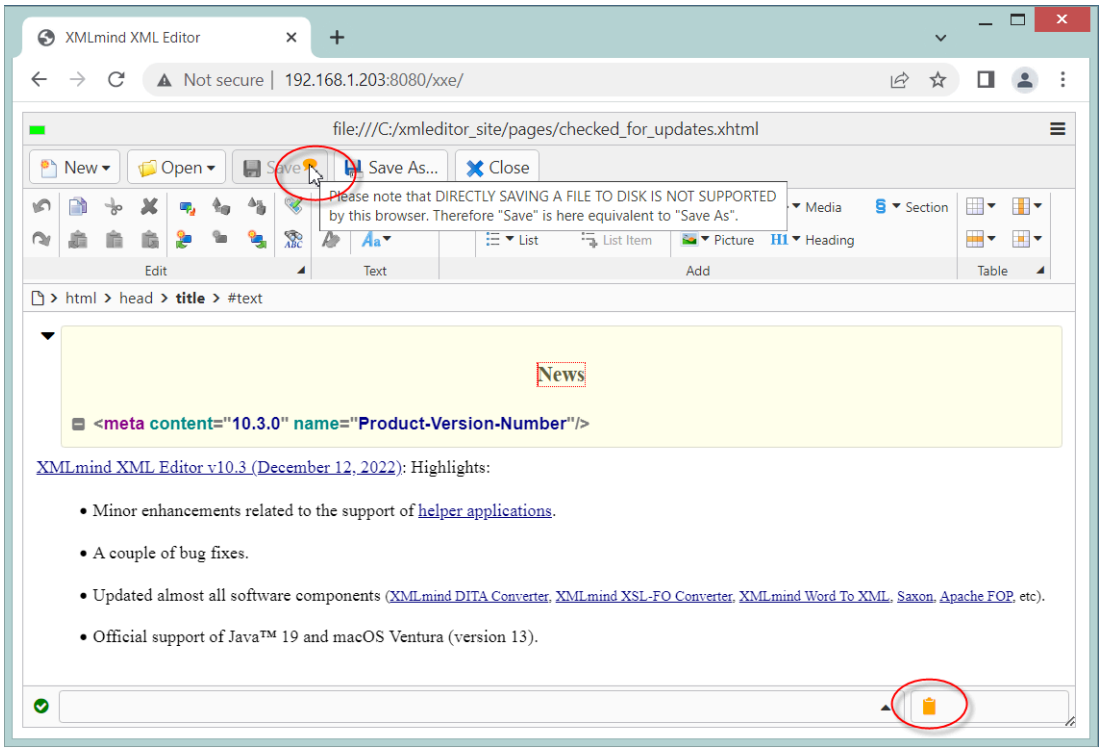
Let's suppose the IP address of `localhost` is `192.168.1.203`. Nothing prevents you from starting a web browser on a different computer and opening HTML page `http://192.168.1.203:18078/xxe/` in it. You'll see the sample XML editor and it will work. However it will not work optimally as features like

- editing local files,
- integrating the system clipboard with the XML editor,

require a *secure context* in order to work.

To make it simple, in order to establish a secure context, the HTML page containing the XML editor must be served over "`http://`**`localhost`**" or "**`https`**`://`" URLs.

*Figure 5-1. The **Save** and **Clipboard** buttons have orange indicators showing you which features are not available in a non-secure context.*



1. Starting `xxeserver` on Linux or on macOS

Let's suppose XMLmind XML Editor Web Edition (**XXEW**) has been installed in `/opt/xxe/` and that SSL certificate `cert_192_168_1_203.pfx` (where `192.168.1.203` is the IP address of your computer) has been copied to `/opt/xxe/web/etc/`.

```
/opt/xxe/web/bin$ nohup xxeserver -pid ../var/xxeserver.pid \
```

```
      -keystore ../etc/cert_192_168_1_203.pfx \
      -storepass changeit -keypass changeit \
      -logserver ../var/srv \
      > /dev/null 2>&1 &
```

- Unix command **nohup** lets you close the terminal used to execute the above command and logout from the computer without shutting down **xxeserver**.

  It would be clearly preferable to deploy **xxeserver** as a *service* but explaining how to do this depends on the operating system used to run **xxeserver** and is out of the scope of this documentation.

- Option "**-pid** ../var/xxeserver.pid" creates text file ../var/xxeserver.pid containing the process ID of **xxeserver**.

  xxeserver can then be stopped as follows:

  ```
  /opt/xxe/web/bin$ kill -SIGTERM `cat ../var/xxeserver.pid`
  ```

  Remember

  Do not forget to delete file ../var/xxeserver.pid otherwise you'll not be able to restart **xxeserver**.

- Options "**-keystore** ../etc/cert_192_168_1_203.pfx **-storepass** changeit **-keypass** changeit" let you specify which SSL certificate to use.

  Any option used to specify an SSL certificate will cause **xxeserver** to establish secure connections. Because option **-port** has not been explicitly used, **xxeserver** URLs will be wss://192.168.1.203:18079/xxe/ws and https://192.168.1.203:18079/xxe/.

- Option "**-logserver** ../var/srv" creates log files related to **xxeserver** as a WebSocket server in directory ../var/srv/ (which will be created if it does not already exist). Such log files should be rather empty because the default value of option **-loglevel** is "WARN,WARN", meaning just log warnings and errors.

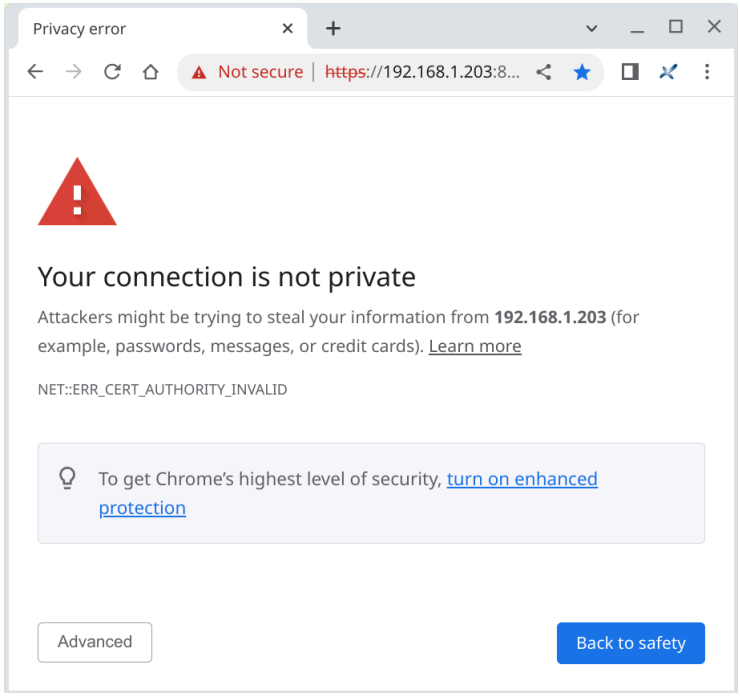
- Option "**-logrequest** ../var/req", not specified in above example, creates log files related to **xxeserver** as an HTTP server in directory ../var/req/. These log files which contains records such as "GET https://192.168.1.203:18079/xxe/index.html" and are rarely useful.

If you don't have an actual SSL certificate, option **-selfsign** lets you quickly generate a self-signed one.

```
/opt/xxe/web/bin$ nohup xxeserver -pid ../var/xxeserver.pid \
      -selfsign "CN=192.168.1.203,O=ACME Corp." ../etc/selfsign_192_168_1_203.pfx \
      -logserver ../var/srv \
      > /dev/null 2>&1 &
```

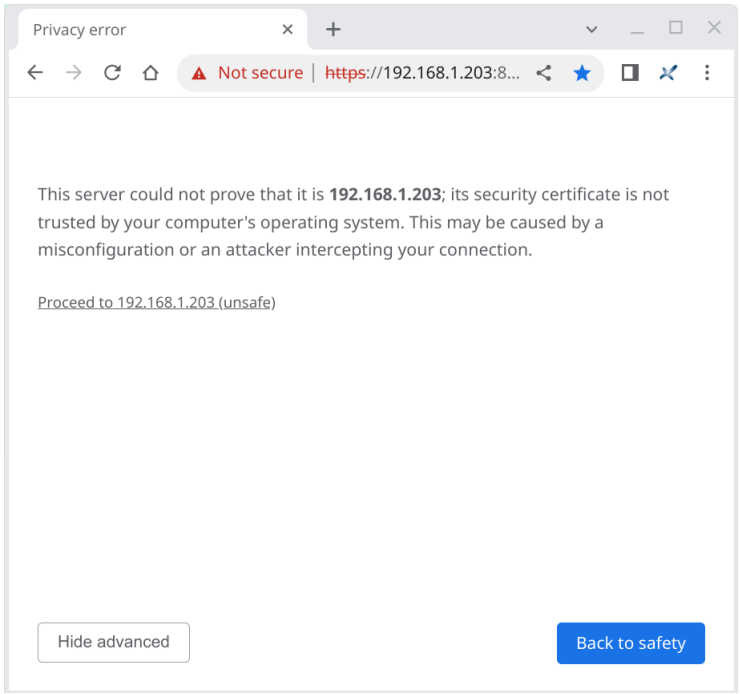
Of course, with a self-signed SSL certificate, all web browsers will report a security issue.

Figure 5-2. Google Chrome reporting a security issue related to an SSL certificate



The user of the web browser will have to click **Advanced** and then "**Proceed to *xxeserver_address* (unsafe)**" to be able to load the HTML page containing **`xxeserver`** client (which is the sample XML Editor in this example). Generally this confirmation must be made just once, the first time you'll load the HTML page containing the client. After that, the web browser will store your self-signed SSL certificate as a “security exception”.

Figure 5-3. Google Chrome letting you accept the self-signed SSL certificate



2. Starting `xxeserver` on Windows

Let's suppose XMLmind XML Editor Web Edition (**XXEW**) has been installed in `C:\xxe\` and that SSL certificate `cert_192_168_1_26.pfx` (where `192.168.1.26` is the IP address of your computer) has been copied to `C:\xxe\web\etc\`.

On Windows, `C:\xxe\web\bin\xxeserver.bat` is of little use as there is no way to keep this command running after you close the Command Prompt used to execute the command and even less, after you log out from the computer.

On Windows, the only way to keep **`xxeserver`** running after you log out from the computer is to install it and start it as a *system service*. This is achieved by using Windows Service Wrapper (**WinSW**), a quality, proven, open source software.

Checking that `xxeserver` works on your computer

Before using `C:\xxe\web\bin\xxeservice.exe` (which is just a renamed `WinSW.exe`), make sure that **`xxeserver`** actually works on your computer. This preliminary step is useful to check the following:

a. Your anti-virus software does not prevent **`xxeserver`** from starting.
b. Windows firewall does not block **`xxeserver`** connections.
c. The port used by **`xxeserver`**, by default 18078 (or 18079 if a SSL certificate has been specified as a command-line option), is available.

Procedure:

1. Open a Command Prompt *as an administrator* and run **`xxeserver`**.

```
C:\xxe\web\bin> xxeserver.bat
```

2. In the address bar of your web browser, type "`http://localhost:18078/xxe/`" then select **New**|**New Local Document** to create a document of any kind and finally click **Close** to close this blank document.

3. Type `Ctrl-C` in the Command Prompt to shutdown **`xxeserver`**.

How to operate `xxeservice`

1. Open a Command Prompt *as an administrator* in order to install and start **`xxeservice`**.

```
C:\xxe\web\bin> xxeservice.exe install
C:\xxe\web\bin> xxeservice.exe start
C:\xxe\web\bin> xxeservice.exe status
```

`install`
: Install the service, that is, register it with Windows service manager.

`start`
: Start the service.

`status`
: Check the current status of the service: `NonExistent` (service not installed), `Started` (service is running) or `Stopped` (service installed but not running).

Remember that `xxeservice.exe` is just a renamed `WinSW.exe`, therefore more information about **`xxeservice`** (that is, **WinSW**) sub-commands is found in *Usage*.

2. In the address bar of your web browser, type "`http://localhost:18078/xxe/`" then select **New**|**New Local Document** to create a document of any kind and finally click **Close** to close this blank document.

3. If you are curious, restart your computer and repeat previous step to check that **`xxeservice`** is still running after the computer is restarted.

4. Open a Command Prompt *as an administrator* in order to stop and uninstall **`xxeservice`**.

```
C:\xxe\web\bin> xxeservice.exe stop
C:\xxe\web\bin> xxeservice.exe status
C:\xxe\web\bin> xxeservice.exe uninstall
```

`stop`
: Stop the service.

`uninstall`
: Uninstall the service.

Actually deploying `xxeservice`

Out of the box, `C:\xxe\web\bin\xxeservice.exe`, whose configuration file is in `C:\xxe\web\bin\xxeservice.xml`, is not very useful. The `<arguments>` element found in this XML configuration file contains the same basic options as those found in `C:\xxe\web\bin\xxeserver.bat`.

```
<arguments>-Xss4m -Xmx2048m -Djava.awt.headless=true
-DXXE_ADDON_PATH="%XXE_ADDON_PATH%" -DXXE_PREFS_DIR="%XXE_PREFS_DIR%"
-classpath "%XXESRVCP%" com.xmlmind.xmleditsrv.server.StartServer
-index "%BASE%\..\webapp\index.html"</arguments>
```

With this configuration:

- The HTML page containing the sample XML editor is `http://localhost:18078/xxe/`. Hence you'll have a *secure context* only if you run the web browser on the same computer as **`xxeservice`**.
- In practice, the sample XML editor only lets you edit *local files*. By default, no matter which user account was used to start **`xxeservice`**, access to *remote files* is limited to the “home directory” of `LocalSystem`, the system account used by the Windows service manager.

The `<arguments>` element which follows contains more useful options[8]:

```
<arguments>-Xss4m -Xmx2048m -Djava.awt.headless=true
-DXXE_ADDON_PATH="%XXE_ADDON_PATH%" -DXXE_PREFS_DIR="%XXE_PREFS_DIR%"
-classpath "%XXESRVCP%" com.xmlmind.xmleditsrv.server.StartServer
-loglevel INFO -logserver "%BASE%\..\var\srv"
-keystore "%BASE%\..\etc\cert_192_168_1_26.pfx" -storepass changeit -keypass changeit
-faccess "%BASE%\..\etc\remote_files_conf.json"
-index "%BASE%\..\webapp\index.html"</arguments>
```

- Variable *`%BASE%`* is predefined by **`xxeservice`** and is substituted with the path of the directory containing `xxeservice.exe` (which is `C:\xxe\web\bin\` in this example).

  Remember that `xxeservice.exe` is just a renamed `WinSW.exe`, therefore more information about the `<arguments>` element, environment variables, etc, is found in *XML configuration file*.

[8] You'll have to edit `C:\xxe\web\bin\xxeservice.xml` using a text or XML editor in order to change the `<arguments>` element.

- By default, the value of option **`-loglevel`** is "`WARN,WARN`", meaning just log warnings and errors. Here, with "`INFO`" (or equivalently "`INFO,WARN`") we want **`xxeserver`** to be a little more verbose.
- Option "**`-logserver`** *`%BASE%`*`\..\var\srv`" creates log files related to **`xxeserver`** as a WebSocket server in directory *`%BASE%`*`\..\var\srv\` (which will be created if it does not already exist).
- Options "**`-keystore`** *`%BASE%`*`\..\etc\cert_192_168_1_26.pfx` **`-storepass`** `changeit` **`-keypass`** `changeit`" let you specify which SSL certificate to use.

  Any option used to specify an SSL certificate will cause **`xxeserver`** to establish secure connections. Because option **`-port`** has not been explicitly used, **`xxeserver`** URLs will be `wss://192.168.1.26:18079/xxe/ws` and `https://192.168.1.26:18079/xxe/`.
- Option "**`-faccess`** `%BASE%\..\etc\remote_files_conf.json`" points to a **JSON** configuration file specifying which remote files may be accessed by **`xxeserver`** client (which is the sample XML Editor in this example). In this example, `remote_files_conf.json` contains just a single line letting the sample XML Editor access any file found in `C:\work`.

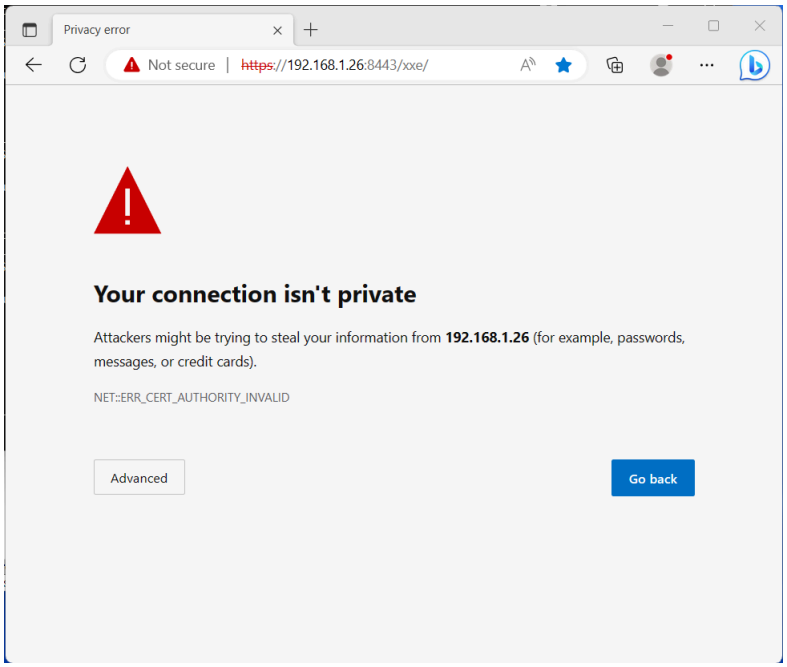
```
[ { "label": "Work", "uri": "file:/C:/work" } ]
```

If you don't have an actual SSL certificate, option **`-selfsign`** lets you quickly generate a self-signed one.

```
<arguments>-Xss4m -Xmx2048m -Djava.awt.headless=true
-DXXE_ADDON_PATH="%XXE_ADDON_PATH%" -DXXE_PREFS_DIR="%XXE_PREFS_DIR%"
-classpath "%XXESRVCP%" com.xmlmind.xmleditsrv.server.StartServer
-loglevel INFO -logserver "%BASE%\..\var\srv"
-selfsign "CN=192.168.1.26" "%BASE%\..\etc\selfsign192_168_1_26.cert"
-faccess "%BASE%\..\etc\remote_files_conf.json"
-index "%BASE%\..\webapp\index.html"</arguments>
```

Of course, with a self-signed SSL certificate, all web browsers will report a security issue.

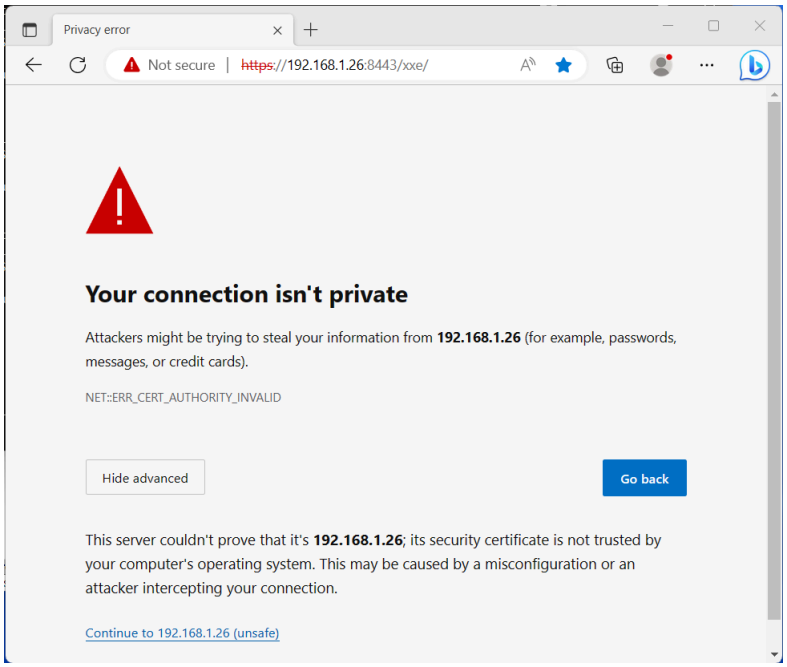
Figure 5-4. Microsoft Edge reporting a security issue related to an SSL certificate



The user of the web browser will have to click **Advanced** and then "**Continue to *xxeserver_address* (unsafe)**" to be able to load the HTML page containing **`xxeserver`** client (which is the sample XML

Editor in this example). Generally this confirmation must be made just once, the first time you'll load the HTML page containing the client. After that, the web browser will store your self-signed SSL certificate as a “security exception”.

Figure 5-5. Microsoft Edge letting you accept the self-signed SSL certificate



Chapter 6. Integrating an XML editor into your web application

1. Overview

Let's assume your web application comprises a frontend running in the user's browser and a backend running on a server computer. Let's call your frontend, **MyFrontend** and your backend, **MyBackend**. **MyFrontEnd** and **MyBackend** communicate with each other through HTTP/HTTPS. **MyFrontEnd** is implemented in HTML/CSS/JavaScript, this code possibly being totally or partially automatically generated by **MyBackend**. **MyBackend** possibly makes use of a database of some sort also running on a server computer.

In order to integrate XMLmind XML Editor Web Web Edition (**XXEW**) into your web application:

- **MyFrontEnd** HTML page must contain `<xxe-client>`, a custom HTML element defined by *JavaScript class* (ECMAScript 6) `XMLEditor`.
- **`xxeserver`**, a WebSocket server, the backend of `<xxe-client>`, must run side by side with **MyBackend**, though not necessarily on the same server computer.

Opening an XML document

1. **MyFrontEnd** JavaScript code queries **MyBackend** to obtain the XML source of the document to be opened in `XMLEditor`.
2. **MyFrontEnd** obtains a “handle” to the instance of `XMLEditor` contained in its HTML page, possibly using `document.getElementById(id)` or `document.getElementsByTagName("xxe-client")`. Let's call this handle `xmlEditor`.
3. **MyFrontEnd** invokes `xmlEditor.openDocument(xmlSource, documentURI)`.

Note

Method `openDocument()` must be passed a document URI identifying the document being edited.

`XMLEditor` makes very few assumptions about how documents are stored by your web application, so your are free to use a URI of any kind, suffice for this URI to be meaningful to your web application.

Using custom URI schemes and/or custom URI authorities is fine ***as long as the document URI is hierarchical***. The syntax of a document URI is thus: *`scheme://authority/path`*, with *`authority`* being optional. For example, the following URIs are supported: `https://cms.acme.com/docs/manual.xml`, `docs:///0943_3561`, and the following URIs are not: `mailto:john@acme.com`, `urn:isbn:9780582035874`.

Creating a new XML document rather opening an existing one is done by invoking `xmlEditor.newDocumentFromTemplate(templateXMLSource, documentURI)`. The main difference with `openDocument` is that after invoking `newDocumentFromTemplate`, the `saveAsNeeded` property of `XMLEditor` is set to `true`.

Saving changes

MyFrontEnd may invoke `xmlEditor.saveDocument()` to save the changes made to the document. Because how documents are stored is entirely the responsibility of **MyFrontEnd**/**MyBackend**, ***this method does nothing at all*** except setting the `saveNeeded` property of `XMLEditor` is set to `false`.

In order to let **MyBackend** ***actually save*** the document being edited, **MyFrontEnd** may invoke `xmlEditor.getDocument()` to first obtain the XML source of the modified document and then send this source to **MyBackend**.

Similarly, `xmlEditor.saveDocumentAs(newDocumentURI)`, which may be used to implement the "**Save As**" command, simply

- changes the `documentURI` property of `XMLEditor` to specified URI,
- sets the `saveNeeded` property is set to `false`,
- sets the `saveAsNeeded` property is set to `false`.

MyFrontEnd almost certainly needs to be informed when changes are made to the document, therefore when these changes need to be saved to the document storage. This is done by registering a "`saveStateChanged`" listener with `XMLEditor` as follows: `xmlEditor.addEventListener("saveStateChanged", listener)`. This listener will receive `SaveStateChangedEvent`s.

Closing the XML document being edited

MyFrontEnd may invoke `xmlEditor.closeDocument()` to close the document being edited, if any.

Several properties of `XMLEditor`, `documentIsOpened`, `documentUID`, `documentURI`, etc, may be used to test whether a document is currently being edited.

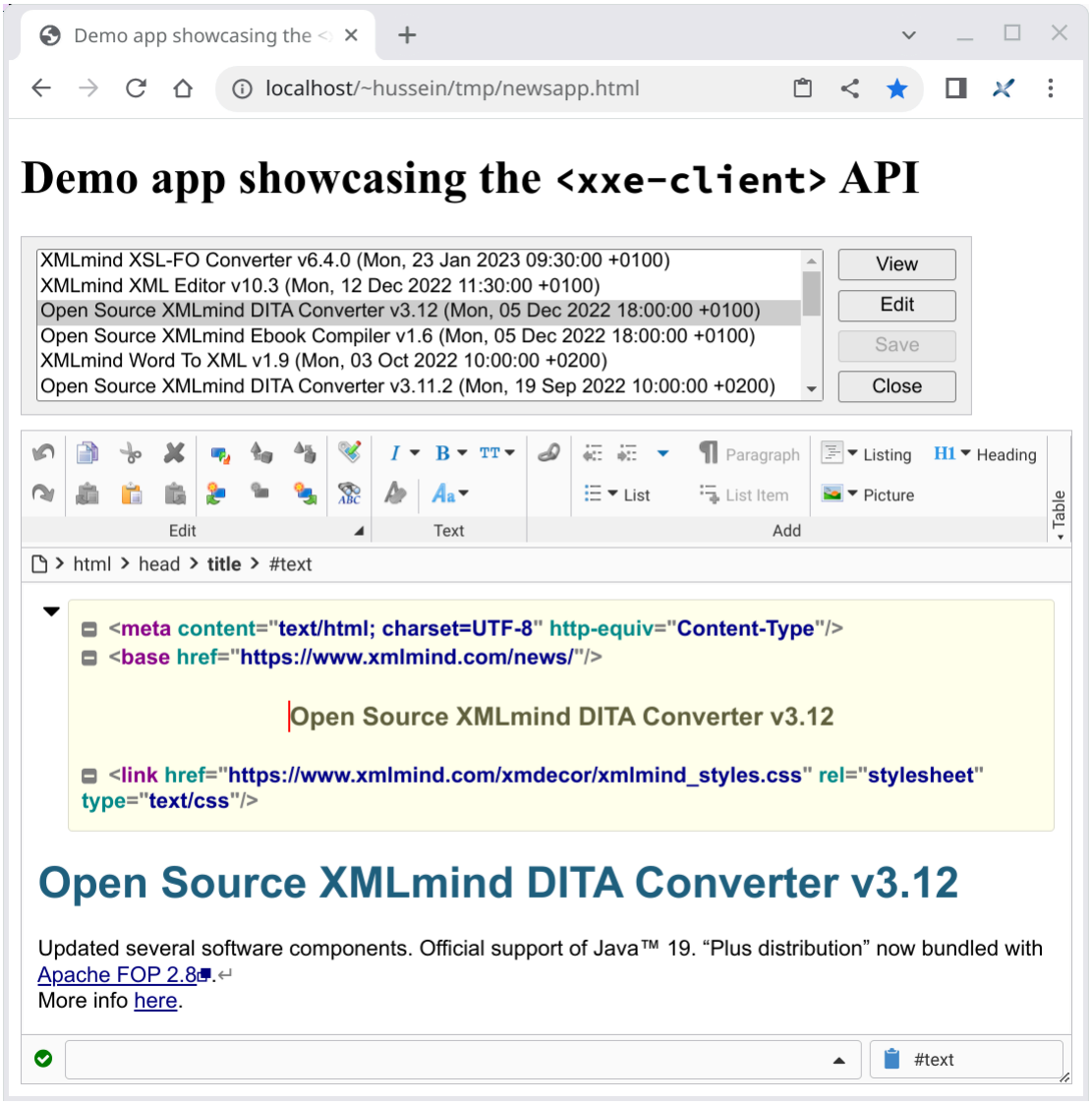
2. Sample web application integrating an XML editor

`XXE_INSTALL_DIR/web/doc/manual/apidemo/` contains `newsapp.html`, `newsapp.js`, `newapp.css`, a sample web application we'll use in this chapter to explain how to integrate `<xxe-client>` (defined by JavaScript class `XMLEditor`) into any other web application.

The **NewsApp** web application mimics a Content Management System (CMS) containing a number of news articles about XMLmind Software products. A news article is a short HTML file. Some news articles have an image attachment. **NewsApp** lets you browse or edit news articles and also “save”[(9)] the changes you made to an article.

[(9)] Previewing the modified news article in a new browser tab is used to simulate saving the document.

Figure 6-1. `newsapp.html` opened in Google chrome; article "DITA Converter v3.12" opened in `<xxe-client>`



In order to mimics a CMS, **NewsApp** loads https://www.xmlmind.com/news/xmlmind.xml, an RSS file containing news items about XMLmind Software products. Each news item simulates a different, standalone HTML document contained in the CMS.

Figure 6-2. `Excerpts from https://www.xmlmind.com/news/xmlmind.xml`

```
<rss version="2.0">
  <channel>
    <title>XMLmind News</title>
    <link>http://www.xmlmind.com/</link>
    ...
    <item>
      <title>Open Source XMLmind DITA Converter v3.12</title>
      <link>http://www.xmlmind.com/ditac/download.shtml</link>
      <description><![CDATA[Updated several
        software components. Official support of Java&trade; 19.
        &ldquo;Plus distribution&rdquo; now bundled with <a
        href="https://xmlgraphics.apache.org/fop/2.8/" target="_blank">Apache
        FOP 2.8</a>.<br />More info <a
```

```
href="http://www.xmlmind.com/ditac/changes.html#v3.12.0">here</a>.]]></description>
      <pubDate>Mon, 05 Dec 2022 18:00:00 +0100</pubDate>
      <guid isPermaLink="true">http://www.xmlmind.com/ditac/changes.html#v3.12.0</guid>
    </item>
    ...
  </channel>
</rss>
```

Running NewsApp

As explained in Section 1. Overview, **`xxeserver`** normally runs side by side with **MyBackend** on a server computer. Therefore the most “realistic” method for running **NewsApp** is:

1. Copy `XXE_INSTALL_DIR/web/doc/manual/apidemo/newsapp.html`, `newsapp.js`, `news.css` and also the whole `XXE_INSTALL_DIR/web/webapp/xxeclient/` to a directory published by your HTTP server.

   For example, on a Linux box having Apache **`httpd`** publishing the contents of *`$HOME/`*`public_html/` directory as `http://localhost/~`*`USER`*`/`, copy all these files to *`$HOME/`*`public_html/tmp/`.

2. Start `XXE_INSTALL_DIR/web/bin/xxeserver`.

   For example, on a Linux box:

   ```
   .../web/bin$ xxeserver
   ```

3. Open `newsapp.html` in a web browser.

   For example, on a Linux box, open `http://localhost/~`*`USER`*`/tmp/newsapp.html`.

Alternatively, if you don't have an HTTP server available for testing **NewsApp**, remember that **`xxeserver`** is not only a WebSocket server but also an HTTP server.

1. Copy `XXE_INSTALL_DIR/web/doc/manual/apidemo/newsapp.html`, `newsapp.js`, `news.css` to `XXE_INSTALL_DIR/web/webapp/`.
2. Start `XXE_INSTALL_DIR/web/bin/xxeserver`.
3. Open `http://localhost:18078/newsapp.html` in a web browser.

NewsApp initialization

An HTML page containing `<xxe-client>` must include `xxeclient/xxeclient.css` and `xxeclient/xxeclient.js` as follows:

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    ...
    <link href="xxeclient/xxeclient.css" rel="stylesheet" type="text/css" />
    <script type="module" src="./xxeclient/xxeclient.js"></script>
    ...
  </head>
  <body>
    ...
    <xxe-client></xxe-client>
    ...
  </body>
```

```
</html>
```

`apidemo/newsapp.js`, being a *JavaScript module* itself, imports everything it needs from JavaScript module `xxeclient/xxeclient.js`. Therefore `apidemo/newsapp.html` does not need to directly include `xxeclient/xxeclient.js`.

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
  ...
  <link href="xxeclient/xxeclient.css" rel="stylesheet" type="text/css" />
  <link href="newsapp.css" rel="stylesheet" type="text/css" />

  <script type="module">//<![CDATA[

import { NewsApp } from "./newsapp.js";

window.onload = (event) => {
    new NewsApp();
}

//]]></script>
  </head>
  <body>
    ...
    <table id="paneLayout">
      <tr>
        <td rowspan="4">
          <select id="itemList" size="6">
            <option value="">Please choose a news item.</option>
          </select>
        </td>
        <td><button type="button" id="viewButton">View</button></td>
      </tr>
      <tr><td><button type="button" id="editButton">Edit</button></td></tr>
      <tr><td><button type="button" id="saveButton">Save</button></td></tr>
      <tr><td><button type="button" id="closeButton">Close</button></td></tr>
    </table>

    <xxe-client id="xmlEditor"
      serverurl="${protocol}://${hostname}:${defaultPort}/xxe/ws"></xxe-client>

  </body>
</html>
```

JavaScript class `NewsApp`, part of JavaScript module `apidemo/newsapp.js`, does all its initializations in its constructor.

```
import * as XUI from './xxeclient/xui.js';
import * as XXE from './xxeclient/xxeclient.js';

...

export class NewsApp {
```

```
    constructor() {
        this._itemList = document.getElementById("itemList");
        this._itemList.disabled = true;
        this._itemList.onchange = this.itemSelected.bind(this);

        this._viewButton = document.getElementById("viewButton");
        this._viewButton.disabled = true;
        this._viewButton.onclick = this.viewItem.bind(this);

        ...INITIALIZE 3 MORE BUTTONS...

        this._xmlEditor = document.getElementById("xmlEditor");
        this._xmlEditor.addEventListener("saveStateChanged",
                                         this.itemSaved.bind(this));

        this._xmlEditor.autoRecover = false;
        window.addEventListener("beforeunload", (event) => {
            if (this._xmlEditor.saveNeeded) {
                event.preventDefault();
                return (event.returnValue = true);
            }
        });

        this._items = [];
        this.loadNews(NewsStorage.baseURI + "xmlmind.xml");
    }

    async loadNews(rssURL) {...}
    ...
    itemSaved(event) {
        this.enableButtons();
    }
}
```

After obtaining a “handle” to `<xxe-client>` (defined by JavaScript class `XMLEditor`) using `document.getElementById`, `NewsApp` configures this instance of `XMLEditor` by invoking method `addEventListener` and by setting property `autoRecover` to `false`.

Remember

The default value of property `autoRecover` is `true`. This means, that by default, the full state of `<xxe-client>` is automatically recovered when the user goes away from the page containing `<xxe-client>`, either intentionally (e.g. the user clicks the "**Reload current page**" button of the browser) or by mistake (e.g. the user closes the web browser tab without saving the changes made to the document).

Having this automatic recovery feature enabled is very reassuring for the user but implies that your web application as whole either have a similar automatic recovery feature or is stateless. The sample XML Editor application, `<xxe-app>`, included in the **XXEW** distribution is stateless and works fine with `xmlEditor.autoRecover=true`.

`NewApp` is also stateless and would work fine with `xmlEditor.autoRecover=true`. However in this `apidemo/newsapp.html` demo, we have chosen to set `autoRecover`

to `false` to explain what to do in the general case. The answer is the `"beforeunload"` event listener found in the above excerpts of `apidemo/newsapp.js`.

Opening a news article

Opening the news article selected in the list is done by invoking `XMLEditor` method `openDocument`. The optional `readOnly` parameter, which is `false` by default, may be used to open an XML document in read-only mode.

Of course before doing that, you must make sure that the user does not unintentionally loose changes made to the news article. This verification/confirmation step is implemented using `XMLEditor` properties `documentIsOpened` and `saveNeeded`.

```
async openItem(readOnly) {
    let sel = this._itemList.selectedIndex;
    if (sel < 0) {
        return;
    }
    const selItem = this._items[sel];

    let confirmed = await NewsApp.confirmDiscardChanges(this._xmlEditor);
    if (!confirmed) {
        return;
    }

    let closed = await NewsApp.closeDocument(this._xmlEditor);
    if (!closed) {
        return;
    }

    let opened = await this._xmlEditor.openDocument(selItem.htmlSource,
                                                    selItem.uri, readOnly);
    if (!opened) {
        return;
    }

    this.enableButtons();
}

static confirmDiscardChanges(xmlEditor) {
    if (!xmlEditor.documentIsOpened || !xmlEditor.saveNeeded) {
        // No changes.
        return Promise.resolve(true);
    }

    return XUI.Confirm.showConfirm(
        `"${xmlEditor.documentURI}" has been modified\nDiscard changes?`);
}
```

Important

As you can see it in the above and following excerpts of `apidemo/newsapp.js`, almost all the methods of `XMLEditor` are *asynchronous* and return a `Promise`. This is why `async` and `await` are used in these excerpts.

Saving a news article after modifying it

A modified news article is not really saved. Clicking the **Save** button just let the user preview the modified news article in a new browser tab. This action is implemented using `XMLEditor` methods `getDocument` and `saveDocument`.

```
async saveItem(event) {
    if (!this._xmlEditor.documentIsOpened || !this._xmlEditor.saveNeeded) {
        return;
    }

    let savedItem = this.findItem(this._xmlEditor.documentURI);
    if (savedItem === null) {
        // Should not happen.
        return;
    }

    const htmlSource = await this._xmlEditor.getDocument();
    if (htmlSource === null) {
        return;
    }
    savedItem.htmlSource = htmlSource;

    let saved = await this._xmlEditor.saveDocument();
    if (!saved) {
        return;
    }
    // No need to enableButtons, there is itemSaved.

    let newWin = window.open("", "_blank");
    newWin.document.write(htmlSource);
    newWin.document.close();
}

findItem(docURI) {
    for (let item of this._items) {
        if (item.uri === docURI) {
            return item;
        }
    }
    return null;
}
```

Closing the news article being viewed or edited

Closing the news article being viewed or edited is done by invoking `XMLEditor` method `closeDocument`. Unless its optional `discardChanges` parameter, `false` by default, is set to `true`, `closeDocument` will not close a document having unsaved changes.

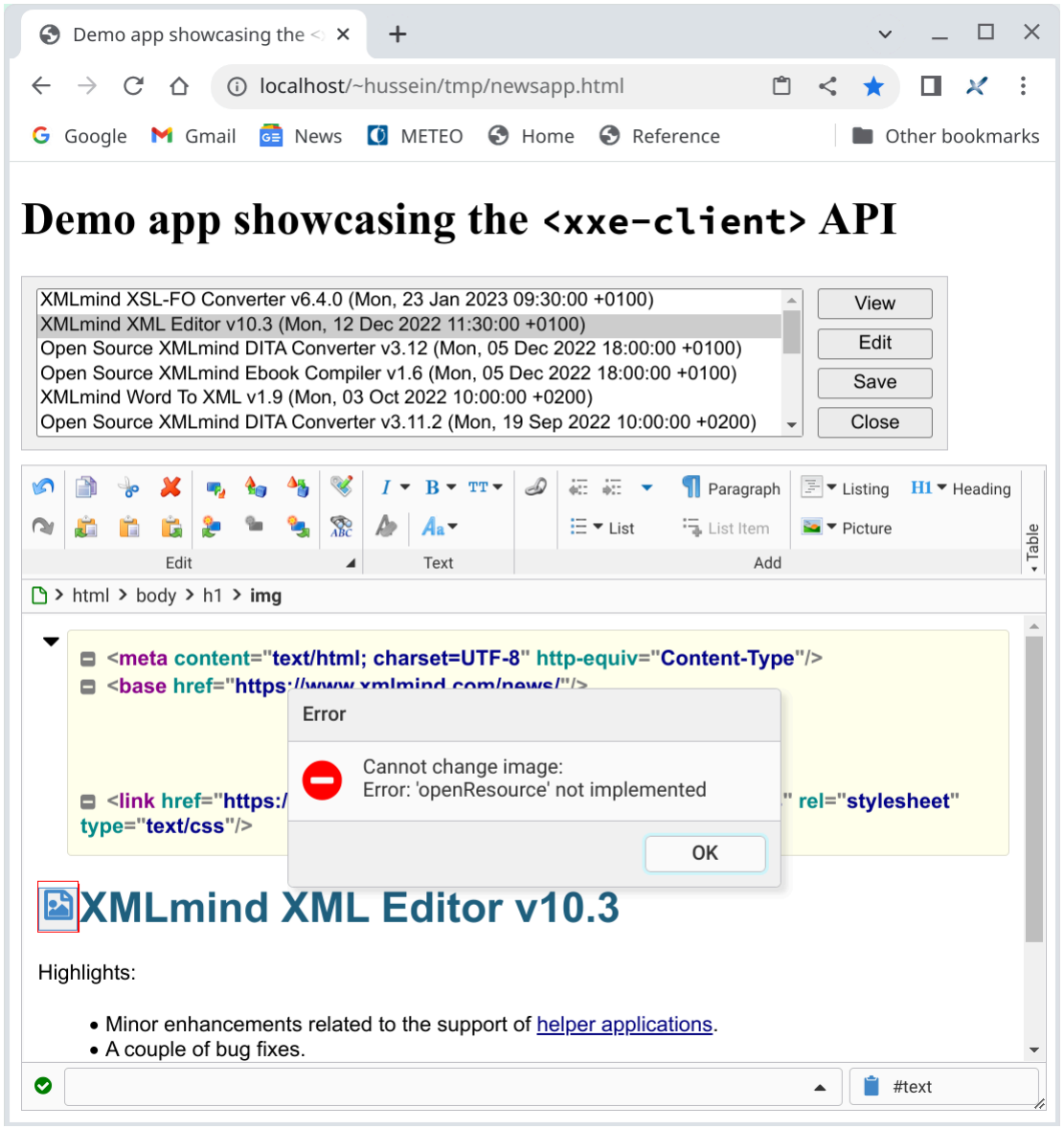
```
static closeDocument(xmlEditor) {
    if (!xmlEditor.documentIsOpened) {
        return Promise.resolve(true);
    }

    return xmlEditor.closeDocument(/*discardChanges*/ true);
}

...

async closeItem(event) {
    let confirmed = await NewsApp.confirmDiscardChanges(this._xmlEditor);
    if (!confirmed) {
        return;
    }

    let closed = await NewsApp.closeDocument(this._xmlEditor);
    if (!closed) {
        return;
    }

    this._itemList.selectedIndex = -1;
    this.enableButtons();
}
```

2.1. Document resources

Let's suppose you want to insert an image into a news article. After using the **Picture** button of the toolbar to insert an `<img>` element and double-clicking (or right-clicking) the image placeholder icon, a dialog box reporting an "*openResource not implemented*" error is displayed. See figure below.

Therefore the only way to specify the `@src` attribute of the newly inserted `<img>` element is to use the **Edit Attributes** dialog box. However, after doing that, the image placeholder icon just changes its color from blue to green and you'll not see the image you have specified.

*Figure 6-3. The **NewsApp** web application without any `ResourceStorage` registered with `XMLEditor`*



This limitation is due to the fact that `XMLEditor` makes very few assumptions about how documents and also document resources like images, video, audio, are stored by your web application.

This limitation may be removed by implementing a `ResourceStorage` and registering it with `XMLEditor` using its `resourceStorage` property.

A `ResourceStorage` object must implement:

`loadResource(`*`uri`*`)`
: Load and return the `Resource` having specified URI.

`storeResource(`*`data, uri`*`)`
: Save resource data (for example, an image `File` dragged from the desktop and dropped onto the image placeholder icon) to specified URI and return the corresponding newly created `Resource`.

`openResource(`*`options`*`)`
: Display a dialog box letting the user choose an existing resource and return the chosen `Resource` object.

A `Resource` is a very simple object essentially associating the resource URI to the resource data (a JavaScript `Blob` or `File`).

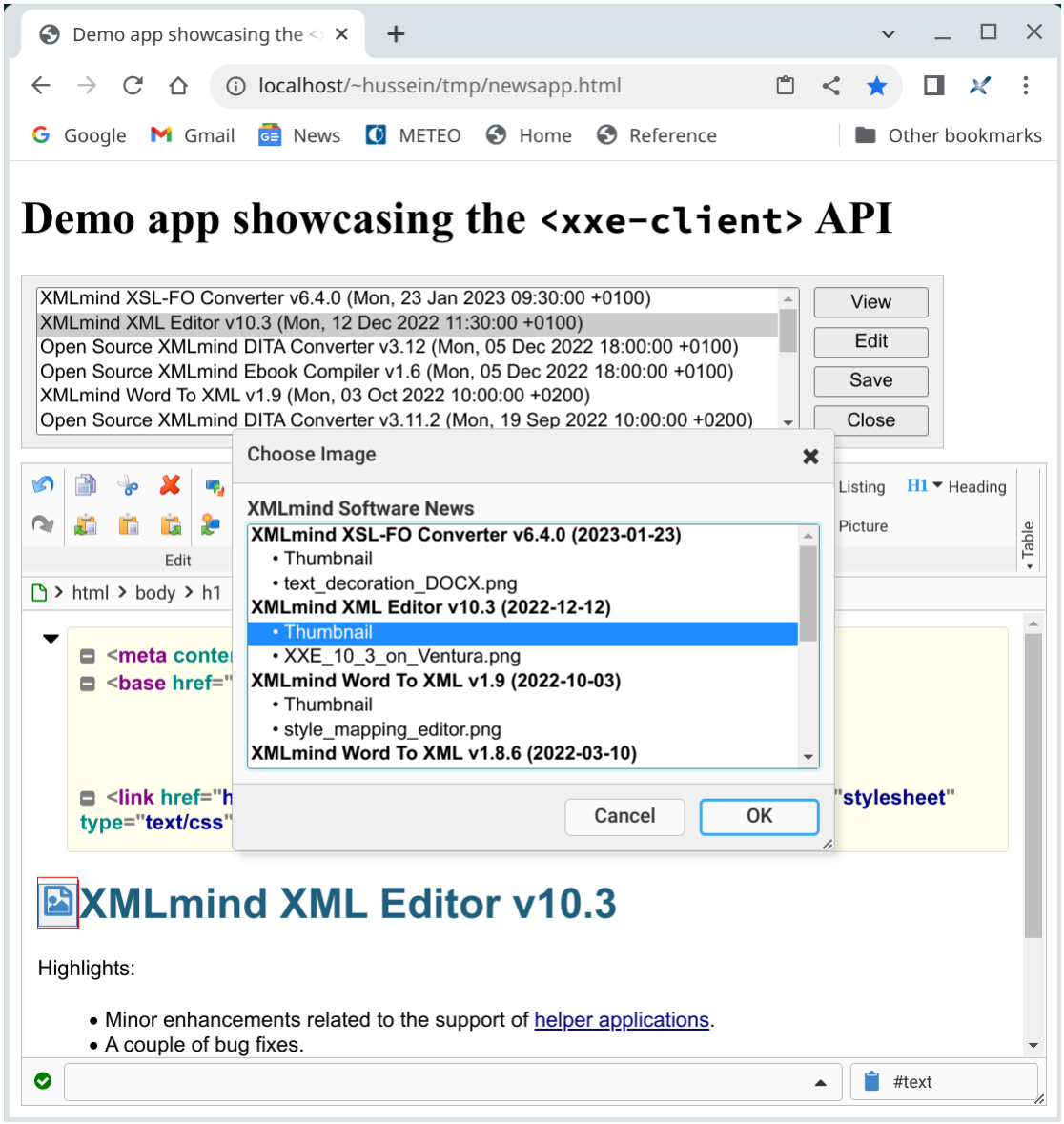
The **NewsApp** web application has a `ResourceStorage` implementation called `NewsStorage` and a `Resource` implementation called `NewsResource`.

Figure 6-4. Excerpts from `apidemo/newsapp.js`

```
class NewsResource extends XXE.Resource {
    constructor(uri, data) {
        super(uri, data);
    }
}

class NewsStorage extends XXE.ResourceStorage {
    constructor(xmlEditor, newsItems) {
        super(xmlEditor);
        this._newsItems = newsItems;
    }

    async loadResource(uri) { ... }

    async openResource(options) {
        let uri = await NewsResourceChooser.showDialog(this._newsItems,
                                                       options);
        if (uri === null) {
            // Canceled by user.
            return null;
        }

        return this.loadResource(uri);
    }
}
...
export class NewsApp {
    ...
    async loadNews(rssURL) {
      ...
      this._items = items;
      this._xmlEditor.resourceStorage =
            new NewsStorage(this._xmlEditor, items);
      ...
    }
    ...
}
```

`NewsStorage.openResource` displays a `NewsResourceChooser` dialog box to let the user choose an image. See figure below.

*Figure 6-5. The **NewsApp** web application having its `NewsStorage` registered with `XMLEditor`*



Chapter 7. `xxeserver` command-line options

`xxeserver`, a WebSocket server, is the backend of XMLmind XML Editor Web Edition (**XXEW**). Its client is custom HTML element `<xxe-client>`.

```
xxeserver [Advanced option]* [Server option]*
```

Advanced options

These options may be used to add, replace or modify some user preferences.

Important

Here the term *user* refers to the user who started **`xxeserver`**, not to the user who is using `<xxe-client>`.

XMLmind XML Editor user preferences are documented in *XMLmind XML Editor - Online Help, Preference keys*. Most user preferences do not apply to the Web Edition (**XXEW**). Examples: *`displayScaling`*, *`useNativeFileChooser`*.

`-putprefs` ***`property_file`***
: Similar to **`-putpref`** except that several key/value pairs may be read from specified property file..

`-putpref` ***`key value`***
: Adds or replace preference specified by key/value to the set of the user's preferences.

`-delpref` ***`key`***
: Removes preference specified by key from the set of the user's preferences..

Server options

`-index` ***`file`***
: Welcome file. Default: *`XXE_INSTALL_DIR`*`/web/webapp/index.html`, *`XXE_INSTALL_DIR`* being the directory where XMLmind XML Editor Web Edition (**XXEW** for short) has been installed. This file contains the sample XML editor application included in **XXEW** distribution.

Remember

This option implicitly sets the *document root* of **`xxeserver`** as an HTTP server. For example, "`-index C:\temp\myapp.html`" sets the document root to "`C:\temp\`". Therefore any file outside "`C:\temp\`" cannot be accessed using an "`http://`" URI.

This also implies that all `<xxe-client>` code (`xxeclient.js`, `xxeclient.css`, etc) must be found somewhere inside "`C:\temp\`" in order to be accessed by **`xxeserver`**.

`-port` ***`port`***
: Port to be used by the server. Default: 18079 if HTTPS, 18078 otherwise. See option **`-keystore`** below.

-keystore *file*
Keystore location. No default. Implies HTTP, not HTTPS.

-storetype *type*
Type of the keystore. Default: `pkcs12` (a `.pfx` file for example).

-storepass *password*
Password for the keystore. No default.

-keypass *password*
Password for the private key. No default.

-certalias *alias*
Alias of the keystore entry. No default.

-selfsign *dname cert_file*
If *`cert_file`* does not already exist, use specified distinguished name *`dname`* to create a self-signed certificate in this file[10]. Then use newly created or existing *`cert_file`* to expose only secure connections to clients. Ignored if option **`-keystore`** is used. No default.

Note

The syntax of distinguished names (*`dname`*) is:

```
CN=cName,OU=orgUnit,O=org,L=city,S=state,C=countryCode
```

- *`cName`* - IP address or fully qualified name of server host
- *`orgUnit`* - department or division name, e.g., '`Purchasing`'
- *`org`* - large organization name, e.g., '`ABCSystems\, Inc.`' (Notice the '`\`' used to protect the '`,`'.)
- *`city`* - city name, e.g., '`Palo Alto`'
- *`state`* - state or province name, e.g., '`California`'
- *`countryCode`* - two-letter country code, e.g., '`CH`'

Each field must appear in the above order but it is not necessary to specify all fields. Examples:

```
CN=192.168.1.203
CN=192.168.1.203,OU=Dev tests,O=ACME Corp.
CN=www.acme.com,O=ACME Corp.,L=San Diego,S=California,C=US
```

Tip

If *`dname`* is "`auto`", then *`cert_file`* may also optionally contain substring "`auto`". In *`dname`*, "`auto`" is replaced by "`CN=`*`IPv4_ADDRESS_OF_THIS_COMPUTER`*" and in *`cert_file`*, "`auto`" is replaced by "`selfsign`*`IPv4_ADDRESS_OF_THIS_COMPUTER`*`.pfx`".

This spares you the effort of determining the IPv4 address of the computer running **`xxeserver`**, which is handy in the case of a quick test. Example, if the IPv4 address of the computer is `192.168.1.26` then "`-selfsign` **`auto`**

[10] If needed to, the parent directories of this file are automatically created too.

`..\etc\auto`" is equivalent to "`-selfsign CN=192.168.1.26 ..\etc\selfsign192.168.1.26.pfx`".

`-loglevel` *`level`*`[,`*`level`*`]?`
: Set logging level to `ALL`, `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR` or `OFF`. Second, optional, level applies to the embedded Jetty server. Default: `WARN,WARN`.

`-logrequests` *`dir`*
: Request logs will be created in this directory. Default: requests not logged.

`-logserver` *`dir`*
: Server logs will be created in this directory. Default: not logged, messages are printed on the console.

`-pid` *`pid_out_file`*
: Write **`xxeserver`** process ID to specified file. Fails if specified file already exists. No default.

  Useful to stop **`xxeserver`** by executing a command equivalent to the Linux example below:

```
kill -SIGTERM `cat pid_out_file`
```

`-faccess` *`config_file`*`|-|~|+|`*`dir_list`*
: Specifies which directories may be accessed by the client.

  - *`config_file`* is **JSON** configuration file specifying which directories may be accessed by the client. **JSON** configuration files are documented in Remote file access.
  - '-' may be used to specify: no file whatsoever.
  - '~' may be used to specify: any file found in the home directory of the user running **`xxeserver`**. Default value.
  - '+' may be used to specify: any file on this computer.
  - *`dir_list`* is a list of absolute or relative directory paths separated by "`;`". Append "`:ro`" to path to make directory read-only. Append "`=`*`label`*" to path to give the directory a label. Example: "`/usr/local/doc:ro;/usr/share/doc:ro=Ref;/home/jjc;/opt/doc=Repo`".

`-maxeditors` *`integer`*
: Maximum number of active XML editors. Default: 25.

`-recoverdocgracetime` *`seconds`*
: Minimum amount of time (in seconds) during which an XML editor may recover its opened document. Default: 300 (5 minutes).

Remote file access

`xxeserver`, the XML editor backend, may be configured to let `<xxe-client>`, the XML editor frontend, access files belonging to its files system. These are called *remote files* as opposed to *local files* which are found in the file system of the computer running the web browser.

The remote file access is specified by the means of a *valid* **JSON** configuration file which is passed to **`xxeserver`** using command-line option **`-faccess`**. The syntax of this **JSON** configuration file is:

```
[
    object [ , object ]*
]

object = {
```

```
"label": label_string ,
"uri": uri_string ,

"readonly": true|false ,

"prompt": prompt_string ,
"scheme": scheme_string ,
"username": username_string ,
"password": password_string

}
```

A **JSON** configuration file contains an array of objects. Each **JSON** object specifies the property of a *remote file root*. `<xxe-client>` may access any file contained directly or indirectly in a remote file root.

JSON object properties are:

`label`
: Required. This label is displayed by the Remote File Chooser. See example below.

`uri`
: Required. The URI of the remote file root. Expected to be an absolute, hierarchical URI ending with '`/`'. May be not only a "`file:///`" URI but also an "`http://`", "`https://`" or "`ftp://`" URI.

  - A remote file root having a "`http://`" or "`https://`" URI requires the "**WebDAV virtual drive plug-in**" add-on to have been installed in XMLmind XML Editor Web Edition (**XXEW**).
  - A remote file root having a "`ftp://`" URI requires the "**FTP virtual drive plug-in**" add-on to have been installed in **XXEW**.

  Tip

  This is best done by running the XMLmind XML Editor desktop application, using menu item **Options**|**Install Add-ons** to download and install this add-on and then starting **`xxeserver`** (which shares its add-ons with the desktop application included in **XXEW** distribution).

  The URI of the remote file root may reference *client properties*. These properties are passed to **`xxeserver`** by the means of the `@clientproperties` attribute of `<xxe-client>` or `<xxe-app>`See example below.

`readonly`
: Optional. Specifies whether the remote file root is read-only or read-write. Read-only means that the user of the XML editor can open files found there but when modified, will have to save them to a different, read-write, remote file root.

`prompt`
: Optional. String used to prompt the user for her/his credentials in order to access a remote file root requiring user authentication. Rarely used.

`scheme`
: Optional. Authentication scheme used to access a remote file root requiring user authentication. Example: "`BASIC`", "`DIGEST`".

Remember

Always use pseudo-scheme "`FTP LOGIN`" when a remote file root has an "`ftp://`" URI.

`username`
: Optional. Username used to access a remote file root requiring user authentication.

`password`
: Optional. Password used to access a remote file root requiring user authentication.

Example:

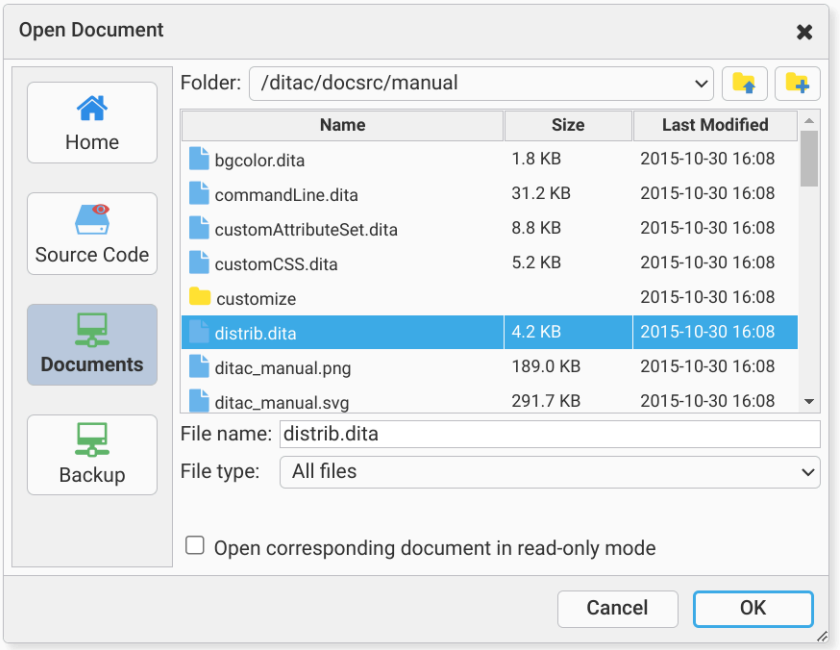
```
[
    { "label": "Home", "uri": "file:///home/~(user)/" },

    { "label": "Source Code", "uri": "file:///usr/local/src/",
      "readonly": true },

    { "label": "Documents", "uri": "http://192.168.1.203/dav/docs/",
      "username": "~(user)", "password": "~(DAV.password)",
      "scheme": "DIGEST" },

    { "label": "Backup", "uri": "ftp://192.168.1.203/backup/",
      "username": "admin", "password": "changeit",
      "scheme": "FTP LOGIN" }
]
```

About above example:

- Variable references `~(user)` and `~(DAV.password)` are substituted with their values. These are *client properties* which are passed to **`xxeserver`** by the means of the `@clientproperties` attribute of `<xxe-client>` or `<xxe-app>`. Example:

  ```
  <xxe-client clientproperties="user=john;group=reviewers\u003Bauthors;DAV.password=change
  ```

- The "**Source Code**" remote file root is read-only.
- The "**Documents**" remote file root requires **WebDAV virtual drive plug-in** to have been installed in **XXEW**. The "**Backup**" remote file root requires **FTP virtual drive plug-in** to have been installed in **XXEW**.
- Labels "**Home**", "**Source Code**", "**Documents**", "**Backup**", are rendered by the Remote File Chooser as follows:



Remote file permissions

In the above example, the remote file root labeled "**Home**" is mapped to `file:///home/~(user)/`. This means that when `<xxe-client>` has been "personalized" with attribute `@clientproperties="user=john"`, **`xxeserver`** will access all files found in `file:///home/john/`. With `@clientproperties="user=jane"`, this will be `file:///home/jane/`, with `@clientproperties="user=jack"`, this will be `file:///home/jack/`, etc.

Let's suppose **`xxeserver`** was started on the server by user *U* belonging to group *G*, this implies that:

- User *U*/group *G* must have sufficient permissions to read and write any file found in `file:///home/john/`, `file:///home/jane/`, `file:///home/jack/`, etc.
- All the files created by **`xxeserver`** in `file:///home/~(user)/` will belong to user *U*/group *G* and not to user `john`, `jane` or `jack`. So what if user `john`, `jane` or `jack` wants to read and/or modify such files using tools other than **XXEW**?

Solving these problems is deemed feasible but depends on the operating system being used to run **`xxeserver`** and is out of the scope of this document.

Tip

On Linux/macOS, a simple solution is to make all users *U*, `john`, `jane`, `jack`, etc, belong to the same group *G* (e.g. `staff`) and to have all the members of this group have an **`umask`** equal to `u=rwx,g=rwx,o=rx`.

Related information

- Chapter 9. The `<xxe-client>` custom HTML element
- Chapter 8. The `<xxe-app>` custom HTML element

1. User preferences

XMLmind XML Editor, the desktop application (**XXE**), stores the user preferences in the following directory:

- *$HOME*`/.xxe10/` on Linux.
- *$HOME*`/Library/Application Support/XMLmind/XMLEditor10/` on the Mac.
- *%APPDATA%*`\XMLmind\XMLEditor10\` on Windows. Example: `C:\Users\john\AppData\Roaming\XMLmind\XMLEditor10\`.

If this user preferences directory —let's call it *XXE_PREFS_DIR*— does not exist, **XXE** automatically creates it and populates it with various sub-directories and files.

xxeserver shares *XXE_PREFS_DIR* with **XXE**. However, there are important differences:

- In the case of **xxeserver**, the "user" is the account which is used to run the server. Therefore different users of `<xxe-client>` cannot have different user preferences.
- **xxeserver** works fine without any user preferences directory and will not automatically create one.
- **xxeserver** will never change the files and sub-directories found in the user preferences directory.

  The **-putprefs**, **-putpref**, **-delpref** command-line options may be used to explicitly override some of the user preferences found in *XXE_PREFS_DIR*`/preferences.properties` and/or the default values of some user preferences, but they will never cause **xxeserver** to modify *XXE_PREFS_DIR*`/preferences.properties`.
- **XXE** and **xxeserver** differ in their use of the sub-directories and files found in the user preferences directory. See table below.

Sub-directory or file	XXE	xxeserver
`preferences.properties`	Java™ property file containing the user preferences. These user preferences are all documented in *Preference keys*.	Most user preferences are ignored as they only apply to **XXE**, the desktop application. However a number of user preferences are considered and may prove to be really useful, for example: • `addOpenLines` and more generally all preferences related to XML formatting when saving a document. • `autoDiffSupport` • `lockLocalDocuments` and more generally all preferences related to file locks. • `makeBackupFiles` • `maxUndo`
`addon/`	Some **XXE** addons may have been installed in this sub-directory.	Add-ons which are not useful in the context of XMLmind XML Editor Web Edition (**XXEW**) are ignored: translation add-ons, spell-checker dictionaries, spell-checker plug-ins, XSL-FO processor plug-ins, any add-

Sub-directory or file	XXE	`xxeserver`
		on in the **Other** category like "**Bidi Support**", "**Edit source**", "**Easy Profiling**", etc. To make it simple, only configuration add-ons are considered.
`cache/`	Serialized (that is, fast-loading) DTDs and W3C XML Schemas may be found in this sub-directory.	Ignored.
`custom/`	Customizations of some **XXE** configurations may be found in this sub-directory.	Ignored.
`spell/`	“Learned words” added by the user to the spell-checker dictionaries may be found in this sub-directory.	Ignored.

Chapter 8. The <xxe-app> custom HTML element

The `<xxe-app>` custom HTML element implements the sample XML editor application included in the XMLmind XML Editor Web Edition distributions.

```
<xxe-app
  autorecover = "false" | "true" : "true"
  autosave = Autosave_specification
  button2pastestext = "false" | "true" : "false"
  checkleaveapp = "false" | "true" : "true"
  clientproperties = Property_list
  documentstorage = "local" | "remote" | "both" : "local"
  serverurl = WebSocket_URL
>
```

Attributes

`@autorecover`

A cover for `<xxe-client>/@autorecover`.

`@autosave`

Specifies which files —local, remote or both— are to be automatically saved and which time interval to use to save them.

The value of this attribute has the following syntax:

```
value = mode [ S interval ]? [ S enabled ]?
mode = local | remote | both
interval = strictly_positive_number s | m | h
enabled = on | off
```

Examples: `"remote"`, `"both 2m"`, `"remote 30s on"`, `"both off"`.

Autosave modes are:

`local`

Automatically save local files (when this is technically possible, i.e. on Chrome, not on Firefox).

`remote`

Automatically save remote files.

`both`

Automatically save both local and remote files.

Autosave interval units are:

`s`

Seconds.

`m`

Minutes.

`h`

Hours.

Default interval is `"30s"`. Minimal interval is `"10s"`.

The default value of *enabled* is "on". This flag specifies whether the autosave feature is *initially* enabled. The user may change this setting at any time using the **Autosave** checkbox found in the **Options** menu.

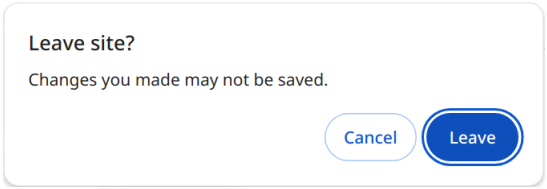
Remember

Unless this attribute is specified, the autosave facility of the sample XML editor application is disabled (the "**Autosave**" checkbox is grayed).

`@button2pastestext`
: A cover for `<xxe-client>/@button2pastestext`.

`@checkleaveapp`
: If set to "`true`", when the document being edited has unsaved changes, ask the user to confirm that she/he really wants to leave the page containing the application. Default value: "`true`".

Figure 8-1. The “leave page” confirmation dialog box of Google Chrome™



`@clientproperties`
: A cover for `<xxe-client>/@clientproperties`.

`@documentstorage`
: Specifies which files `<xxe-app>` can access:

  `local`
  : Default value. `<xxe-app>` can access files found on the computer running the web browser. These are called *local files*.

  `remote`
  : `<xxe-app>` can access found on the computer running **`xxeserver`**. These are called *remote files*.

    Which remote files may be accessed and how these files are accessed —read-write or read-only— may be configured in **`xxeserver`**. See Chapter 7. **`xxeserver`** command-line options.

  `both`
  : `<xxe-app>` can access both local and remote files.

Remember

It's not possible to **Save As** a local file as a remote file. It's not possible to **Save As** a remote file as a local file.

`@serverurl`
: A cover for `<xxe-client>/@serverurl`.

JavaScript API

The <xxe-app> custom HTML element is defined as follows:

```
window.customElements.define("xxe-app", XMLEditorApp);
```

The JavaScript API of class XMLEditorApp is found here.

Related information

- Chapter 7. **xxeserver** command-line options
- Chapter 9. The <xxe-client> custom HTML element

Chapter 9. The <xxe-client> custom HTML element

`<xxe-client>`, a custom HTML element, is the frontend of XMLmind XML Editor Web Edition. It's a client of backend **`xxeserver`**.

```
<xxe-client
  autoconnect = "false" | "true" : "true"
  autorecover = "false" | "true" : "true"
  button2pastestext = "false" | "true" : "false"
  clientproperties = Property_list
  serverurl = WebSocket_URL
>
```

Attributes

`@autoconnect`

Default: `true`. If `true`, automatically connect to **`xxeserver`** when creating a new document, opening a document, etc.

`@autorecover`

Default: `true`. If `true`, automatically recover opened document when the user moves away from the XML editor without closing the document being edited. This automatic document recovery happens for example when:

- the user clicks the "**Go back**" button of the browser and then clicks "**Go forward**";
- the user clicks the "**Reload current page**" button of the browser;
- the user closes and then reopens the browser tab/window containing the XML editor.

`@button2pastestext`

Default: `false`. If `true`, selecting text by dragging the mouse automatically copies this text to a dedicated private clipboard. Then clicking button #2 (middle button) elsewhere pastes copied text at the clicked location. This allows to emulate the *X Window Primary Selection* on all platforms.

Note that the X Window Primary Selection is ***not*** *natively supported* on platforms where it should be (e.g. Linux) because it seems there is no way to update the Primary Selection without updating the System Clipboard at the same time.

`@clientproperties`

Default: no client properties. Specifies a number of property name/property value pairs which are typically used to associate the user of `<xxe-client>` with the **`xxeserver`** connection (`<xxe-client>` *peer*; see Part I, Chapter 2. How it works). On the server side, these client properties are seen by the `<xxe-client>` peer as Java™ system properties, which makes them usable in different contexts (macros, access to remote file systems, etc).

Note that client properties may be used only to *add new system properties* and not to override existing system properties. For example, specifying client property `user.name=john` will not cause standard system property `user.name` to be modified on the server side.

This attribute is almost always set by some third-party JavaScript code used to integrate **XXEW** with the information system of this third-party (e.g. a CMS). Example:

```
user=john;group=reviewers\u003Bauthors;DAV.password=changeit
```

The syntax of this attribute is:

```
properties = property [ ';' property ]*
property = name '=' value
```

If value contains ';', this character may be escaped as '\u003B'. See above example: group value is "reviewers;authors".

@serverurl

Specifies the "ws://" (WebSocket) or "wss://" (WebSocket Secure) URL of **xxeserver**. Default: "*${protocol}*://*${hostname}*:*${port}*/xxe/ws", where *${protocol}*, *${hostname}* and *${port}* represent variable values computed using the URL of the HTML page containing <xxe-client>.

Supported variables are:

Variable reference	Substituted value
${protocol}	"wss" if <xxe-client> loaded from an "https://" URL; "ws" otherwise.
${hostname}	Same host name or IP address as in the URL of the HTML page containing <xxe-client>.
${port}	Same port as in the URL of the HTML page containing <xxe-client>, knowing that implicit HTTPS port is 443 and implicit HTTP port is 80.
${defaultPort}	18079 if <xxe-client> loaded from an "https://" URL; 18078 otherwise.

Simple examples demonstrating how the default value of @serverurl is computed:

- <xxe-client> found in http://localhost:18078/xxe/index.html, @serverurl is ws://localhost:18078/xxe/ws.
- <xxe-client> found in https://www.xmlmind.com/xmleditor/_web/demo/index.html (implicit HTTPS port is 443), @serverurl is wss://www.xmlmind.com:443/xxe/ws.

JavaScript API

The <xxe-client> custom HTML element is defined as follows:

```
window.customElements.define("xxe-client", XMLEditor);
```

The JavaScript API of class XMLEditor is found here.

Related information

- Chapter 7. **xxeserver** command-line options
- Chapter 8. The <xxe-app> custom HTML element

Part III. Using XMLmind XML Editor Web Edition

Learn how to use **XMLmind XML Editor Web Edition** (**XXEW** for short). Desktop Application users should feel at home with **XXEW** to a very large degree and may want to skip reading this part of the document.

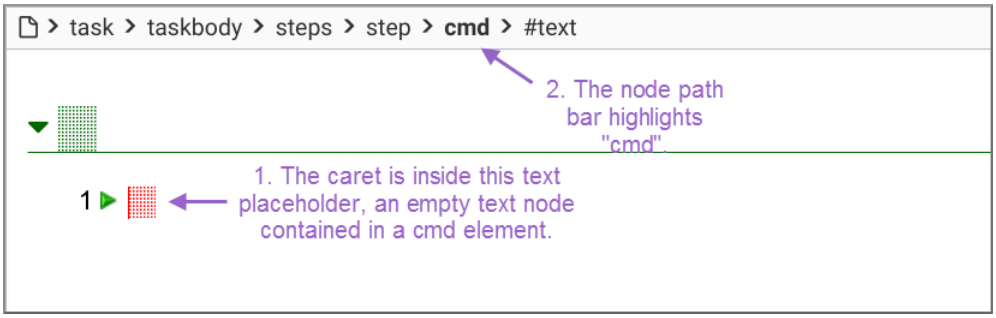
Chapter 10. The basics

A few things you need to learn before starting to use XMLmind XML Editor Web Edition (**XXEW** for short).

XXEW is quite straightforward to use but you simply cannot guess how it works. While what you see resembles your typical word processor, **XXEW** does not work like a word processor, nor like a text editor, neither like other XML editors. Therefore this chapter is really a must read. (Corresponding 5min screencast.)

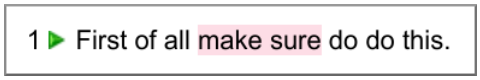
Basic concepts

- Patterns looking like this  are *text placeholders*. You can click into (or tab to) such placeholders and start typing.
- The *node path bar*, found above the document view, indicates what is selected or when there is no explicit selection, the element containing the caret.

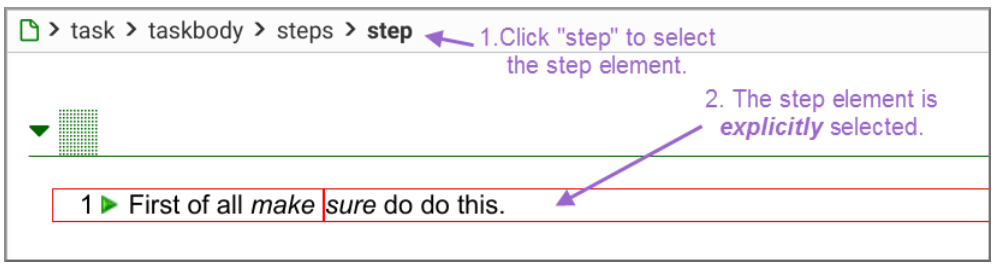


- **XXEW** has 2 different kinds of selection, the *text selection* and the *node selection*.

  a. The text selection, which is given a pink background color, works like in any word processor or text editor. Few editing commands apply to the text selection: **Paste**, **Convert**. For example, you can convert the text selection to a bold or italic inline element.

  b. The node selection looks different than the text selection: a thin red frame is drawn around the selected nodes. The simplest way to select a node is to click on its name in the node path bar.

  Most editing commands apply to the node selection.

  The node selection may comprise several sibling nodes, for example, two contiguous paragraphs. The easiest way to extend the node selection is to `Shift-click` before or after the thin red frame.

- The element directly containing the caret is always *implicitly selected*: no thin red frame around it.

  For example if you want to insert a table before a paragraph, first click inside this paragraph (but not inside any of its child elements, as this would implicitly select the child element) then use **Insert Before**.

  However, if you want to insert a list after a table, you cannot do that using the implicit element selection because a table cannot directly contain some text. You'll have to first explicitly select this table (e.g. by clicking on "`table`" in the node path bar) then use **Insert After**.

- Notice that the two editing commands mentioned above are **Insert Before**, **Insert After**. **XXEW** also has less commonly used **Insert** editing command.

  Insert Before
  : Insert an element (or a text node) *just before the node selection*.

  Insert
  : Insert an element *right here, at caret position*.

  Insert After
  : Insert an element (or a text node) *just after the node selection*.

  Note

  Note that ***XXEW** does not work like other XML editors*. In other XML editors,

  – **Insert Before** often means "Insert *somewhere* before selection".
  – **Insert** often means "Insert *somewhere* inside selection".
  – **Insert After** often means "Insert *somewhere* after selection".

- For the same reasons, **XXEW** has 3 paste commands and not just one: **Paste Before**, **Paste**, **Paste After**. Unlike **Insert**, **Paste** is commonly used and works as expected by replacing the text or node selection with the contents of the clipboard.

- When **XXEW**, which is a strictly validating XML editor, does not allow you to perform the editing command you want, it's almost always because you didn't select the right element and/or you are not using the right editing command.

  For example, you have clicked inside a paragraph and attempt to use **Insert** to add a section after it. This cannot work because a paragraph cannot contain a section. Instead, first select the section which is the ancestor of the paragraph (e.g. click "`section`" in the node path bar) then use **Insert After** (not **Insert**) and select "`section`" from the list.

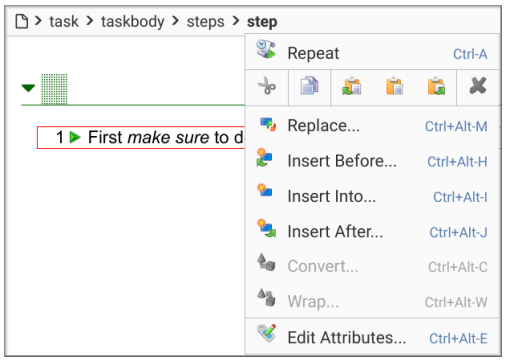
Basic editing commands

What are the basic editing commands and where to find them?

The contextual menu of the node path bar

In **XXEW**, most basic editing commands are invoked by selecting an entry of the contextual menu of the node path bar. While clicking an element name or node type (e.g. "`#text`") in the node path bar just explicitly selects this element or node, *right-clicking* not only selects this element or node but also displays a contextual menu containing all the basic editing commands.

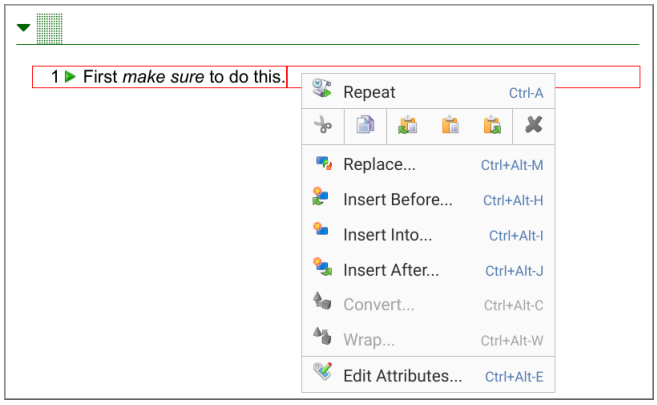
Figure 10-1. The contextual menu of the node path bar



The contextual menu of the text or node selection

Right-clicking inside the text selection (having a pink background color) or explicit node selection (inside the thin red frame) displays a contextual menu containing all the basic editing commands.

Figure 10-2. The contextual menu of the text or explicit node selection

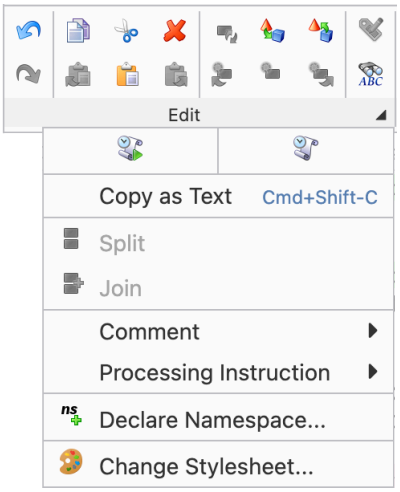


Note that right-clicking in the document view when there is no text or explicit node selection displays the contextual menu of the web browser. This contextual menu is only useful for fixing a spell-checking mistake using the spell-checker of the web browser.

The toolbar

The left side of the toolbar is fixed and contains all the editing commands which are generic, that is, not specific to a given document type (e.g. DocBook, DITA Topic, XHTML). Moreover clicking ◢, a small button found at the left of the **Edit** label, displays a menu which contains even more generic commands.

Figure 10-3. The left side of the toolbar and its popup menu

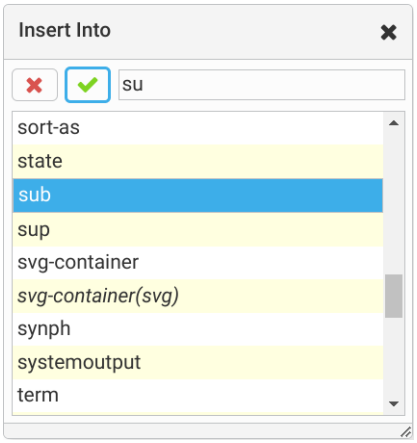


Commands **Undo**, **Redo**, **Copy**, **Cut**, **Delete** are found in all editors and will not be described here. Commands **Paste Before**, **Paste**, **Paste After** and commands **Insert Before**, **Insert**, **Insert After** have already been discussed in the previous section.

Replace

Replace the node selection by a new, empty, element or text node. A dialog box is displayed to let the user choose this new element or text node ("`#text`").

*Figure 10-4. The same element chooser dialog box is used for the **Insert Before**, **Insert**, **Insert After**, **Replace**, **Convert** and **Wrap** commands*



Convert

Replace the text or node selection by an element containing this text or node selection. Example 1: select a paragraph and use **Convert** to convert it to a program listing. Example 2: make a text selection mixing text and inline elements and use **Convert** to convert it a bold inline element.

Wrap

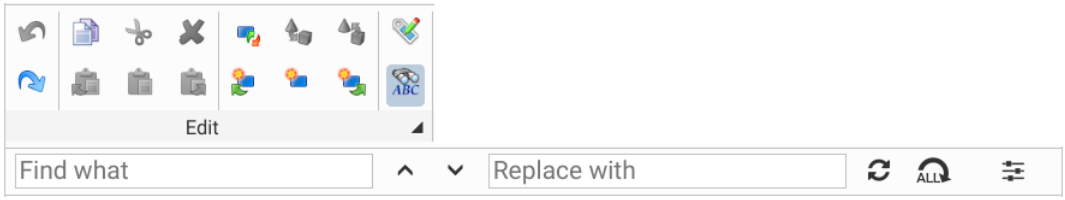
This command is a variant of **Convert**. The only difference between **Wrap** and **Convert** is that, with **Wrap**, when a single element is selected, the selected element is

given a new parent element. Example[11]: select a paragraph and use **Wrap** to give it a note parent.

Edit Attributes
Displays a dialog box which may be used to add, remove and change the attributes of implicitly or explicitly selected element.

Search/Replace
Expands/collapses the text search/replace pane which is part of the toolbar.

*Figure 10-5. The text search/replace pane is revealed after clicking "**Search/Replace**"*



Repeat
Repeats last repeatable command. See Repeat some of the commands you have already executed.

Command History
Displays a dialog box listing last repeatable commands from newest to oldest.

Copy as Text
Copies as *plain text* —just the characters, not the elements— the explicit text or node selection to the clipboard.

Split
Splits explicitly selected element in two parts, the split point being specified by caret position.

Tip

Commands **Split** and **Join** are rarely used because for most document types pressing `Enter`, `Backspace` and `Delete` mimic the behavior corresponding keystrokes in a word processor and thus may be used to split and join elements. Examples:

- Pressing `Enter` inside a paragraph or a list item splits this element in two parts.
- Pressing `Backspace` at the beginning of a paragraph or a list item joins this element to the preceding paragraph or a list item.
- Pressing `Delete` at the end of a paragraph or a list item joins this element to the following paragraph or a list item.

---

[11] In this example, we'll assume that case a note must contain paragraphs hence a paragraph may not be converted to a note.

Join

Joins explicitly selected element to its preceding sibling, an element of same type. This gives a single element containing the child nodes of the two joined elements. This command is the inverse command of **Split**.

Comment sub-menu

Sub-menu containing commands which may be used to insert a comment node at caret position and to insert a comment before or after selected node.

Processing instruction sub-menu

Sub-menu containing commands which may be used to insert a processing instruction node at caret position, to insert a processing instruction before or after selected node and to change the target of a processing instruction.

Declare Namespace

Displays a dialog box letting the user declare a namespace, change the *prefix* of a namespace or make a namespace the *default namespace*.

Note

If the current document is conforming to a DTD, the dialog box lets the user view the namespaces and their prefixes but not modify them.

Change Stylesheet

Displays a dialog box letting the user choose an alternative CSS stylesheet for the styled view or on the contrary, no stylesheet at all, that is, switch to the tree view.

Adding images to your document

1. Use **Insert Before**, **Insert**, **Insert After** or, easier, the **Picture** button often found in the right side of the toolbar to add an element representing an image to your document.

   The image element will be inserted into your document but at first, you'll only see , the blue *image placeholder* icon.

Note

An image placeholder icon is given different colors and different tooltips in order to explain its presence:

Blue

Image file not yet specified.

Green

Image file specified but could not be displayed, either because the image format is not supported (e.g. EMF) or because the document being edited was opened from a local file (see note about local images).

Red

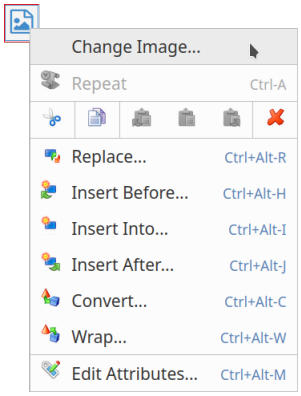
An error occurred when attempting to display the image. The image tooltip contains an error message.

2. Specify an image file. There are 3 different methods to do this.

Double click or right-click the image placeholder icon

The quickest way to do this is to double click the image placeholder icon (or the image itself if an image file has already been specified) or or right-click image placeholder icon and select the "**Change Image**" entry from the contextual menu. This invokes the "**Change Image**" command which changes the image file of an image element.

*Figure 10-6. the "**Change Image**" menu entry*



a. You'll first be prompted to choose an image file. The image file chooser being displayed by the "**Change Image**" command depends on whether the document being edited was opened from a local file or from a remote file.

Figure 10-7. The local image file chooser dialog box

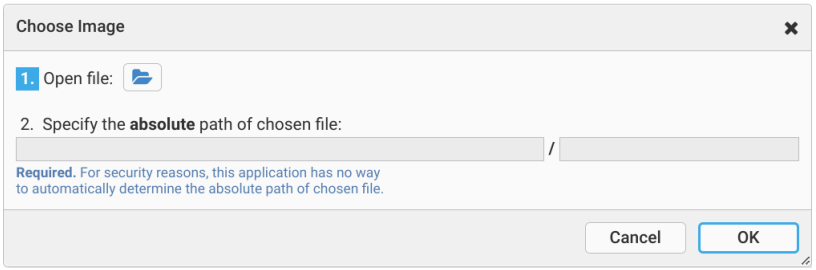
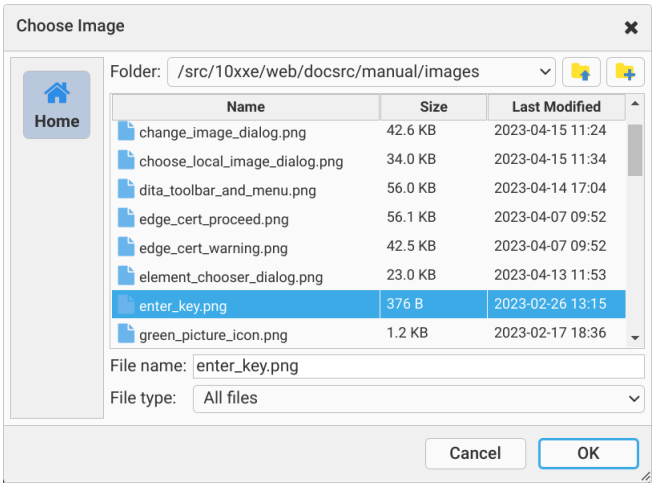


Figure 10-8. The remote image file chooser dialog box



b. You'll then be prompted to specify whether the chosen image file is to be referenced by the image element or to be embedded[12] in the document.

Figure 10-9. The "Change Image" dialog box

Alternatively, drop an image file onto the image placeholder icon

Alternatively, drag an image file and drop it onto the image placeholder icon (or the image itself if an image file has already been specified). This also invokes the "**Change Image**" command, sparing you the effort of choosing an image file using a dialog box.

Note

When the document being edited was opened from a remote file, this method is the only way to add to your document an image coming from a local file.

Alternatively, specify the attribute of the image element pointing to the image file

Alternatively:

a. Click inside the image placeholder icon (or the image itself if an image file has already been specified) to select the corresponding image element. The node path bar will show you the name of this element. This image element depends on the type of the document being edited: DITA Topic `<image>`, DocBook `<imagedata>`, XHTML `<img>`.

[12] Not recommended for document size and possible interchange problem reasons.

b. Use **Edit Attributes** to specify the attribute pointing to the image file. This attribute depends on the type of the document being edited: DITA Topic `<image>/@href`, DocBook `<imagedata>/@fileref`, XHTML `<img>/@src`.

Note

When the document being edited was opened from a local file, there is no way to display an image file specified this way. This has already been explained in note about local images. However, after using this method, the blue image placeholder icon will turn to green.

Related information

- Chapter 11. Being productive

Chapter 11. Being productive

Previous chapter may have given you the impression that **XXEW** is straightforward to use but pretty low-level. This is not the case. **XXEW** has most of the facilities found in word processors making the user more productive at editing documents. (Corresponding 3min25 screencast.)

Quickly type some text

You can type text only if the caret is inside a *textual node* (text, comment or processing instruction nodes). Press `Tab` to move the caret to the following textual node. Press `Shift-Tab` to move the caret to the preceding textual node.

Press `Ins` (`F1` on the Mac) to move the caret to the text node found after the element currently containing the caret. If there is no such text node then add a new empty one, that is, add a text placeholder .

If you want to type some text *before* the element currently containing the caret, use `Shift-Ins` (`Shift-F1` on the Mac) instead of `Ins`.

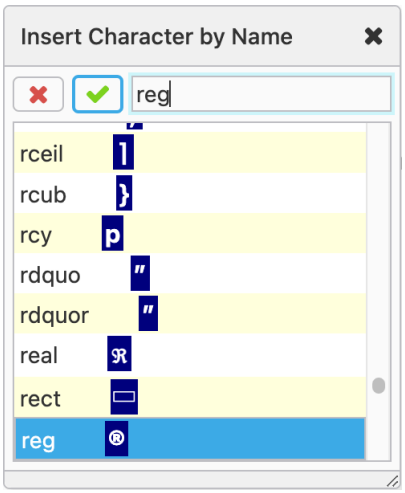
Tip

The `Ins` (`F1` on the Mac) keyboard shortcut is especially useful when you are typing some text inside a bold or italic inline element and now want to end this inline element by typing plain text after it.

Insert special characters

- Press `Ctrl-SPACE` to insert a non-breaking space character (unicode `U+00A0`).
- Type `Esc n` (that is, type `Esc` then type `n`) to display the dialog box letting you choose and insert a special character by its name. Common special characters are: `ldquo` “, `rdquo` ”, `trade` ™, `reg` ®, `mdash` —.

Figure 11-1. The "Insert Character By Name" dialog box



Quickly select an element

- `Ctrl-mouse-click` (`Cmd-mouse-click` on the Mac) selects the node clicked upon. If you continue to `Ctrl-mouse-click` without moving the mouse, this selects the parent of currently selected node and so on until the root element of the document has been selected.
- Pressing `Ctrl-ArrowUp` (`Cmd-ArrowUp` on the Mac) selects the textual node containing the caret. Pressing `Ctrl-ArrowUp` again selects the parent of currently selected node and so on until the root element of the document has been selected. Press `Ctrl-ArrowDown` (`Cmd-ArrowDown` on the Mac) to move the selection down the node hierarchy.
- Click the bullet or the number of a list item to select the corresponding list item element. More generally if the view of an element has a “decorative label” of some kind, clicking this label selects the corresponding element.

Repeat some of the commands you have already executed

Most commands which prompt the user to choose an argument from a list are made *repeatable*. For example, command **Insert After** displays a dialog box letting you choose an element name or "`#text`" (a text node) from a list. Once executed, there is a way to repeat exactly the same **Insert After** command elsewhere in the document without having to display the element choosers dialog box.

- Pressing `Ctrl-A` (`Cmd-A` on the Mac) repeats the execution of last repeatable command, and this, as always, if and only if this is allowed by the DTD or schema of the document given the current editing context.
- Pressing `Ctrl+Shift-A` (`Cmd+Shift-A` on the Mac) displays a dialog box letting you choose a repeatable command from a list in case you want to repeat the execution of a command other than the last one.

The commands corresponding to the above keyboard shortcuts are **Repeat** and **Command History**. These commands are are both found in the "**Edit**" menu of the toolbar.

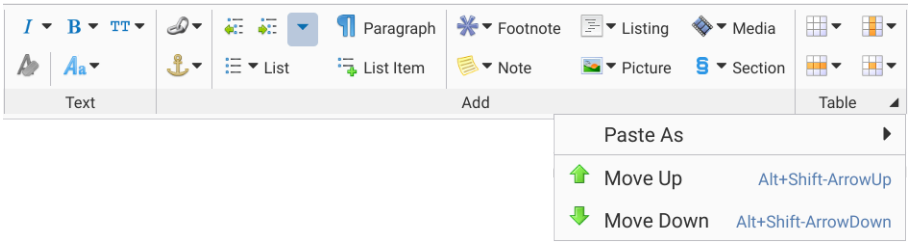
Quickly add the same element

- Pressing `Enter` at the very end of a paragraph or list item adds a new paragraph or list item after current one. Pressing `Enter` at the very beginning of a paragraph or list item adds a new paragraph or list item before current one.
- Pressing `Ctrl-Enter` (`Cmd-Enter` on the Mac) anywhere inside a paragraph or list item adds a new paragraph or list item after current one. Pressing `Ctrl+Shift-Enter` (`Cmd+Shift-Enter` on the Mac) anywhere inside of a paragraph or list item adds a new paragraph or list item before current one.
- Pressing `Ctrl-Ins` (`Esc s` on the Mac; that is, type `Esc` then type `s`) in implicitly or explicitly selected element adds a new element of the same type after selected element. Pressing `Ctrl+Shift-Ins` (`Esc S` on the Mac; that is, type `Esc` then type `S`) in implicitly or explicitly selected element adds a new element of the same type before selected element.

Use as much as possible the commands found in the right side of the toolbar

The right side of the toolbar depends on the type (e.g. DocBook, DITA Topic, XHTML) of the document being edited and contains many commands which are convenient to use. Moreover clicking ◢, a small button found at the bottom/right of the toolbar, displays a menu which contains even more convenient commands.

Figure 11-2. The right side of the "DITA Topic" toolbar and its popup menu



- The "**Text**" section contain commands like **Toggle Italic Inline Element**, **Toggle Bold Inline Element**, **Convert to Plain Text**, etc, which are similar to those found in the toolbars of most word processors.
- The "**Add**" section contain commands like **Paragraph**, **List Item**, **Picture**, etc, which, unlike **Insert Before**, **Insert**, **Insert After** do not require you to be precise in first implicitly or explicitly selecting an element before executing the command. Instead, such commands adds elements after the node selection or after the caret at a location where it is *valid* to do so and where it *makes sense*[13] to do so.

Related information

- Chapter 10. The basics

[13] DITA example: even if the content model of a DITA `<p>` element allows a `<p>` to contain a `<table>`, the new `<table>` element will be added by **Add Table** somewhere after selected `<p>` and never *inside* selected `<p>`.

Appendix A. How to adapt an existing ".xxe" configuration file to XXEW

Like **XXE**, the desktop app, **`xxeserver`** scans the *`XXEW_install_dir`*`/addon/` and *`XXE_user_preferences_dir`*`/addon/` add-ons directories during its startup and load configurations from there.

Unlike **XXE**, **`xxeserver`** automatically skips certain configuration elements (`<binding>`s other than keyboard bindings, `<attributeVisibility>`, `<elementVisibility>`, `<documentSetFactory>`), certain configuration files ("**MathML support**", "**XMLmind XML Editor Configuration Pack**", etc) and certain add-on categories ("spell checker dictionaries", "spell checker plug-ins", "XSL-FO processor plug-ins", etc) because these are either not useful in the context of **XXEW** or because these are not yet supported by **XXEW**. Moreover, **XXE** features are not enabled in **`xxeserver`**. For example, the `ConvertDocument` feature is not enabled, therefore `<xxe-client>` has no "**Convert Document**" submenu.

Restriction

There is currently no way to force a running **`xxeserver`** to reload one or all of its configurations.

Therefore you can very easily make **XXEW** reuse existing configurations created for **XXE** or if you want to create a configuration for **XXEW**, simply follow the instructions which apply to **XXE**.

There is one big restriction though: interactive commands, that is, commands written in Java™ displaying Java dialog boxes, won't work in **XXEW**.

Note

What happens in `<xxe-client>` if the user invokes a interactive Java command which has not yet been “ported” to JavaScript™? Not much. If the command is found in a menu or toolbar, the corresponding menu item/toolbar button will generally be disabled and the user will not be able to invoke the command. At worse, if the user manages to invoke the command, the command will do nothing at all.

Fortunately there is a simple way to mark parts of a configuration file as being specific to **XXEW** or, on the contrary, as being specific to **XXE**. This way some configuration elements are used only when the file is loaded by **`xxeserver`** and other configuration elements are used only when the file is loaded by the desktop app.

Conditional processing of configuration files

It's strongly recommended to use the following processing-instructions to mark parts of a configuration file as being specific to **XXEW** or, on the contrary, as being specific to **XXE**:

```
<?if TEST?>
...configuration elements...
<?else?>
...configuration elements...
```

```
<?endif?>
```

- *TEST* is XXE_CLIENT for configuration elements which are specific to **XXEW** and !XXE_CLIENT for configuration elements which are specific to **XXE**.
- The <?else?> directive is optional.

Use this facility to mark

- interactive command declarations,
- macro command definitions invoking interactive commands,
- menu items, toolbar buttons, bindings invoking interactive commands,
- and more generally any functionality which is not useful in the context of **XXEW**

as being specific to **XXE**.

DocBook examples (excerpts from *XXEW_install_dir*/addon/config/docbook/docbook.xxe):

```
<?if !XXE_CLIENT?>
...
<command name="docb.editImageMap">
  <class>com.xmlmind.xmleditext.docbook.EditImageMap</class>
</command>
<?endif?>

<command name="{docb}setLinkEnd">
  <macro>
    <sequence>
      <test context="$implicitElement" expression="is-editable()" />

      <set variable="selectedElement" context="$implicitElement"
           expression="(ancestor-or-self::*[@%0])[last()]" />

      <?if XXE_CLIENT?>
      <command name="stop"
               parameter="xxeClientExecuteCommand editAttributes %0" />
      <?else?>
      <command name="putAttribute" parameter="%0" />
      <?endif?>
    </sequence>
  </macro>
</command>

<menu label="_DocBook">
  <?if !XXE_CLIENT?>
  <item label="Upgrade to DocBook _version 5..."
        command="docb.toV5"/>
  <separator/>
  <item label="_Set up olinks..."
        command="docb.olinkedDocuments"/>
  <separator/>
  <?endif?>
  ...
</menu>
```

Adapting an interactive macro to XXEW

In some cases, a menu command or a macro command ending with an interactive command can be easily ported to **XXEW** by the means of the `stop` command.

The `stop` command is specific to **XXEW** and does not exist in **XXE**. It's a very simple command which stops the execution of the macro and returns to its invoker a `STOPPED` status and its parameter as the result of the command.

In the above `{docb}setLinkEnd` macro, "`putAttribute` *`attribute_name`*", which is an interactive Java command, has been replaced by "`stop xxeClientExecuteCommand editAttributes` *`attribute_name`*".

By convention, when `<xxe-client>` invokes a remote command (here it's `{docb}setLinkEnd`) and this command stops and returns a result which starts with "`xxeClientExecuteCommand`", then `<xxe-client>` invokes the command specified in this result.

In above example, `editAttributes` is an interactive command written in JavaScript which is the **XXEW** equivalent of interactive Java command `editAttributes`.

Other DocBook example:

```
<command name="{docb}linkMenuItems2">
  <menu>
    <?if !XXE_CLIENT?>
    <item label="Follow Link"
          command="start" parameter="helper(defaultViewer) '%*'" />
    <?else?>
    <item label="Follow Link" command="stop"
          parameter="xxeClientExecuteCommand openExternalLink %*" />
    <?endif?>
    <item label="Set Link Target..."
          command="{docb}setLinkEnd" parameter="url" />
  </menu>
</command>
```

“Porting” an interactive Java command to XXEW

Reading the following section should give you an idea on how difficult it is to “port” an interactive Java command to **XXEW**. It's by no mean a detailed, step by step, tutorial.

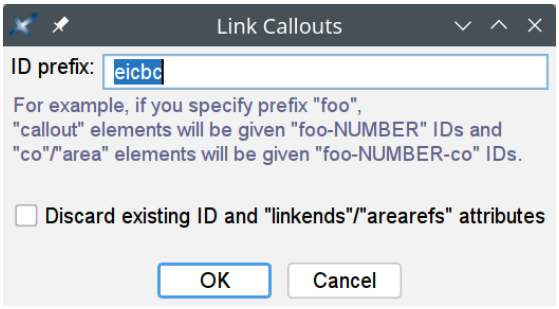
Let's use DocBook command `LinkCallouts` as an example. This interactive Java command, found in the **DocBook** menu, links a sequence of `<callout>` elements to the corresponding sequence of `<co>` or `<area>` elements (and, of course, also the other way round).

▶ First of all, the server-side, interactive Java command must be made “portable” to **XXEW**. This is the case of `LinkCallouts` because:

- The command may be used interactively as well as non-interactively.

  When passed an ID prefix as its parameter, `LinkCallouts` does its work and modify the document being edited without having to display its Java dialog box.

Figure A-1. The Java dialog box displayed by command `LinkCallouts` when used interactively



- The command can be executed on computers having no display (typically when the command is invoked by **`xxeserver`**[(14)]).

  `LinkCallouts` tests whether it can display its Java dialog box. When displaying a dialog box is needed and this is not possible, instead of just failing, `LinkCallouts` returns a `STOPPED` status and the result of the command contains all the information needed to populate the dialog box it would have displayed.

```
Component dialogParent = docView.getDialogParent();
...
if (dialogParent == null) {
    // Cannot prompt user.
    return CommandResult.stopped(stoppedValue(
        prefix, discardExistingXRefs, lockDiscardExistingXRefs));
} else {
    // Prompt user for an ID prefix.
    PrefixDialog dialog = new PrefixDialog(dialogParent);
    Object[] result = dialog.getPrefix(prefix, discardExistingXRefs,
                                       lockDiscardExistingXRefs);
    if (result == null) {
        return CommandResult.CANCELED;
    }
    ...
```

▶ The second effort consists in implementing a client-side, interactive JavaScript command, displaying a dialog box written in HTML+CSS+JavaScript, having the same registered command name as its Java counterpart.

This client-side command is implemented by JavaScript class `LinkCalloutsCmd` and it is declared to `<xxe-client>` as being `docb.linkCallouts` (for use by DocBook 4 documents) and `db5.linkCallouts` (for use by DocBook 5+ documents). Excerpts from *`XXEW_install_dir`*`/web/webapp/xxeclient/docbook.js`:

```
class LinkCalloutsCmd extends XXE.InteractiveRemoteCommand {
    constructor(commandName) {
        super(commandName);
    }

    resumeExecution(mode, docView, params,
```

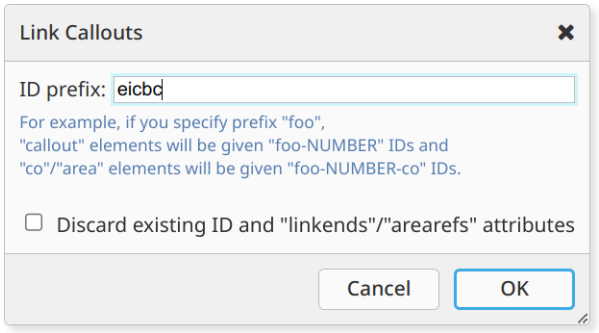
(14) **`xxeserver`** is designed to run on computers having no display hence **`xxeserver`** is started with `-Djava.awt.headless=true`.

```
                    stoppedCommandInfo, resolve, reject) {
        // stoppedCommandInfo syntax is:
        // 'discard'|'keep' [ '!' ] [ ';' prefix ]

    ...
}

for (let n of [ "docb.linkCallouts", "db5.linkCallouts" ]) {
    XXE.ALL_LOCAL_COMMANDS[n] = new LinkCalloutsCmd(n);
}
```

Like all `XXE.InteractiveRemoteCommand`s, `LinkCalloutsCmd` functions as follows:

1. It invokes the remote, that is, server-side, Java, command having the same name (e.g. `docb.linkCallouts`, implemented by Java class `LinkCallouts`) without any parameter.
2. After receiving the result of the remote command (in method `resumeExecution()`), normally a `STOPPED` status and a result value containing all the information needed to populate a dialog box (an ID prefix, if any, and other settings in the case of `docb.linkCallouts`), `LinkCalloutsCmd` displays its HTML+CSS+JavaScript dialog box.

   Figure A-2. The HTML+CSS+JavaScript dialog box displayed by command `LinkCalloutsCmd`

3. Unless the user cancels this dialog box, `LinkCalloutsCmd` invokes one more time remote command `docb.linkCallouts`, but this time with a parameter containing the ID prefix and the other settings specified by the user in the dialog box.
4. Remote command `docb.linkCallouts` having all needed information to do its job, modifies the document accordingly and returns an `DONE` result, which is returned as is as the result of `LinkCalloutsCmd`.

Related information

- Part I, Chapter 2. How it works

Appendix B. Troubleshooting

Troubleshooting: `xxeserver` does not start

- An error message similar to the following one is displayed in the terminal or Command Prompt used to start **`xxeserver`** or is found in **`xxeserver`** log file.

```
xxeserver: cannot start server: Failed to bind to 0.0.0.0/0.0.0.0:18078
```

  Possible causes:

  – **`xxeserver`** is already running.
  – OR the port 18078 is used by another server. On Linux, command "`lsof -i :`*`PORT_NUMBER`*" will tell you which server is currently listening to port *PORT_NUMBER*.

Troubleshooting: the sample XML editor web application does not work

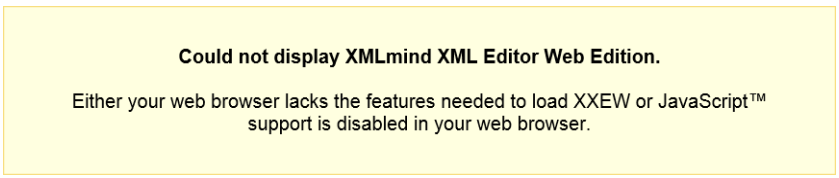
- When opening the HTML page containing the sample XML Editor, you see the following error message:

  Possible causes:

  – The sample XML editor web application loads fine but your web browser is really not supported (e.g. Safari, any mobile web browser).

    You need to switch to a *very recent version of Google Chrome* or to any browser using the same Blink browser engine: Edge, Opera, Brave, etc. Firefox works fine too, but without system clipboard integration.

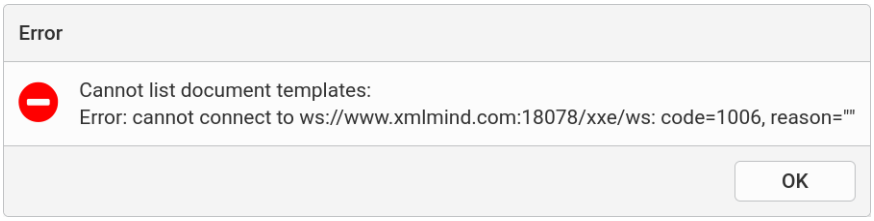
- When opening the HTML page containing the sample XML Editor, you see the following error message:



  Possible causes:

  – The sample XML editor web application did not load. You are using a somewhat obsolete web browser (e.g. Internet Explorer) or your web browser is supported but you have disabled its JavaScript support.

- Opening the HTML page containing the sample XML Editor seems to work but using the **New** or **Open** button displays the following error dialog box:



Possible causes (assuming that **`xxeserver`** and the sample XML editor web application are correctly configured):

- **`xxeserver`** is not running.
- OR the port used by **`xxeserver`** (18078 in the above screenshot) is blocked by your anti-virus, firewall or proxy.

Appendix C. History of changes

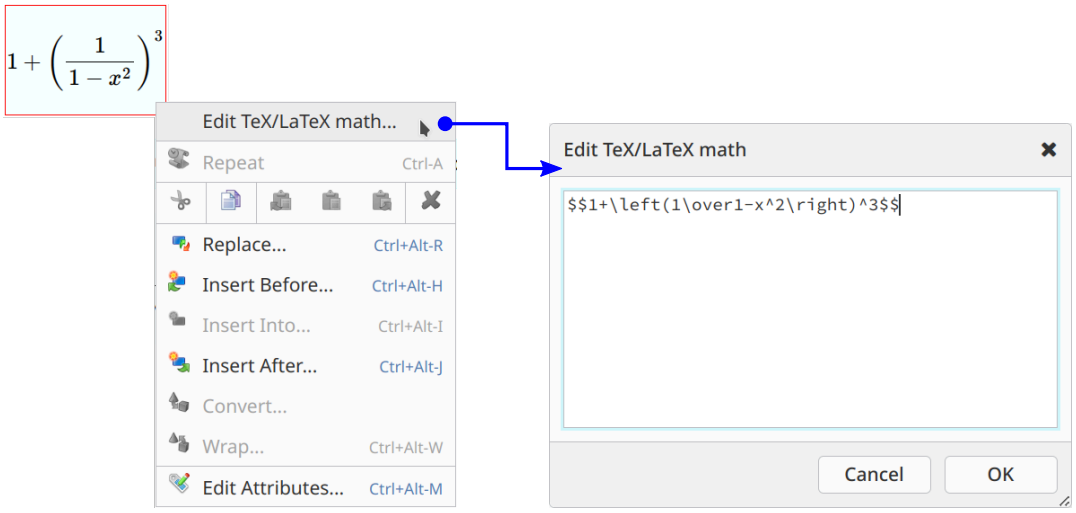
v1.6.0 (May 2, 2025)

`xxeserver` based on XMLmind XML Editor v10.11.

Enhancements:

- Just like the desktop app:
  - DITA and DocBook documents may now embed TeX/LaTeX math or reference “image files” containing TeX/LaTeX math (".tex" files).
  - Shift-clicking the **Copy** tool bar button (or the **Copy** icon found in the contextual menu) now copies the implicit or explicit selection as plain text. This is equivalent to selecting item "**Copy as Text**" in the **Edit** menu of the toolbar.
  - New CSS proprietary extension: `inline-tree`, a new value for the `display` property.

  More information in v10.11 changes.
- MathML and TeX/LaTeX math are now rendered by MathJax, which is dynamically loaded by `xxe-client` when needed to.
- Double-clicking on MathML (respectively TeX/LaTeX math) or right-clicking and choosing "**Edit MathML**" (respectively "**Edit TeX/LaTeX math**") from the contextual menu opens a very simple editor letting you modify this math.



- Right-clicking an image (respectively a remark) now displays the full contextual menu with "**Change Image**" (respectively "**Edit Remark**") added as its first entry.
- **`xxeserver`**: upgraded nanojson to version 1.9.
- **`xxeserver`** embeds Jetty in order to implement an HTTP and WebSocket server. Upgraded Jetty to version 11.0.25.

Bug fixes:

- Custom controls corresponding to remarks, `command-button`, `value-editor`, did not take into account the fact that a document could be opened in read-only mode.

v1.5.0 (November 15, 2024)

`xxeserver` based on XMLmind XML Editor v10.10.

Enhancements:

- Extending the right-click, contextual popup menu by the means of commands found in "`contextualMenuItems`" namespaces (as always, just like in the desktop app) is now supported. For example, if you right-click a DITA `<xref>` element, whether in the document view or in the node path bar, two new menu items "**Follow Link**" and "**Set Link Target**" are added to the “normal” contextual popup menu (containing **Repeat**, **Cut**, **Copy**, ..., **Edit Attributes**).
- **`xxeserver`** embeds Jetty in order to implement an HTTP and WebSocket server. Upgraded Jetty to version 11.0.24.
- Added appendix "*How to adapt an existing "`.xxe`" configuration file to **XXEW***" to "*XMLmind XML Editor Web Edition - Manual*".

Bug fixes:

- Generated content inherited box-related style properties from their parent which, in some cases, caused the rendering to be slightly incorrect. Example, excerpts from `docbook51/css/assembly.css`:

```
instance,
instance[linking] {
    display: block;
    content: icon(left-link, navy)
             text(attr(linkend), font-family, monospace, color, navy)
             " " invoke("attributeValues", linking);
    margin-left: 4ex;
}
```

  The generated content for the above `text()` was also given `margin-left:4ex` creating a large gap between the link icon and the link text.

v1.4.0 (September 3, 2024)

`xxeserver` based on XMLmind XML Editor v10.9.

Enhancements:

- Improved the usability of the diagnostics pane displayed by the "**Check Document Validity**" button by rendering error messages in a nicer, more readable, way.
- Added a "**Show Content Model**" item to the menu of `<xxe-app>`, the sample XML editor application. Just like in the desktop application, this menu item opens a dialog box containing an automatically generated reference manual listing all the elements and attributes specified in the DTD, W3C XML Schema or RELAX NG schema of the document being edited.
- Content object `attributes` (CSS proprietary extension) generating an attributes pane in now supported.
- **`xxeserver`** embeds Jetty in order to implement an HTTP and WebSocket server. Upgraded Jetty to version 11.0.22.

Bug fixes:

- Firefox only: empty (e.g. newly created), styled, having a block or inline-block display, comments and processing-instructions were rendered as a red line.
- In content objects generating form controls (CSS proprietary extensions) like `gauge`, a parameter like "`min, attr(min)`" was not supported.

- In content objects generating form controls (CSS proprietary extensions) like `radio-buttons`, a empty attribute value was not visible. It is now rendered as "`(empty string)`".
- CSS system font values like "`message-box`", "`caption`", etc, were not correctly rendered.

v1.3.0 (June 17, 2024)

`xxeserver` based on XMLmind XML Editor v10.8.

Enhancements:

- When the document being edited has unsaved changes, the sample XML editor application (`<xxe-app>`) now asks the user to confirm that she/he really wants to leave the page containing the application. This feature corresponds to new attribute `<xxe-app>/@checkleaveapp`. Default attribute value: "`true`".

  Note that the document auto recovery feature is orthogonal to the above feature. The document auto recovery feature in the sample XML editor application corresponds to new attribute `<xxe-app>/@autorecover`. Default attribute value: "`true`".

- Added the "**Comment Out**" and "**Uncomment**" menu items to the **Edit**|**Comment** menu of the toolbar. "**Comment Out**" replaces text selection or explicit node selection by a comment containing the selection. "**Uncomment**" is inverse command of "**Comment Out**". It parses the content of implicitly or explicitly selected comment and replaces this comment by parsed XML nodes.

  This feature is a “port” from the desktop app. See "*Comment menu*".

- Added the "**Remark**" submenu to the **Edit** menu of the toolbar. A remark is simply a `<?xxe-remark?>` processing-instruction nicely rendered as a “balloon” in the styled view. Double-clicking on this balloon displays a remark editor. This remark editor may be used to create a new remark or to reply to/modify an existing remark.

  This feature is a “port” from the desktop app. See "*Remark menu*".

- All the content objects generating form controls (CSS proprietary extensions) are now supported. For example, `check-box`, `date-picker`, `radio-buttons`, `value-editor` are supported.

  However there a few limitations compared to the form controls supported by the desktop app:

  - `file-name-field` is rendered like `text-field`. There is no button next to the text field which opens a dialog box letting you choose a file.
  - The following properties of `gauge` are ignored: `low`, `high`, `optimum`, `low-color`, `high-color`, `optimum-color`.
  - The following properties of `spinner` are ignored on the client-side: `pattern`, `language`, `country`, `variant`, `columns`. That is, on the client-side, it's the web browser which is in control of what the user can see and type.
  - The following properties of `date-picker`, `time-picker`, `date-time-picker` are ignored on the client-side: `format`, `pattern`, `language`, `country`, `variant`, `columns`. That is, on the client-side, it's the web browser which is in control of what the user can see and type.

Bug fixes:

- The `autorecover` feature did not work when several documents were opened in several browser tabs/windows.
- Firefox only: pasting some text into a element containing a large number of text lines (e.g. a large `<pre>`) sometimes caused the caret to be scrolled out of sight.

v1.2.0 (March 22, 2024)

xxeserver based on XMLmind XML Editor v10.7.

Enhancements:

- The DITA map, DocBook assembly and Ebook toolbars now have "**Open Topic R/O**" (open topic in read-only mode) and "**Open Topic**" buttons. For example, in the case of a DITA `<map>` or `<bookmap>`, these buttons open in a new browser tab the topic or sub-map referenced by selected `<topicref>`.
- DITA, DocBook, XHTML, TEI Lite configurations: `Ctrl-Alt-click` (`Option-Cmd-click` on the Mac) on an external `http`/`https` link now opens the corresponding page in a new tab. In previous versions of **XXEW**, this action only followed *internal* links.
- **xxeserver** embeds Jetty in order to implement an HTTP and WebSocket server. Upgraded Jetty to version 11.0.20.
- **xxeserver** **-faccess** option: in addition to *config_file*|-|~|+, this option now also supports *dir_list*, where *dir_list* is a list of absolute or relative directory paths separated by ";".
- The `xxe-web-eval-N_N_N` distribution is now available as a Docker image called `xmlmind/xxe-web:N_N_N-eval`. How to run this image as a container is documented in its Docker Hub page.

Bug fixes:

- When navigating to a folder containing files having very long names, the "**Open Remote Document**" dialog box automatically became wider, which was somewhat annoying.

v1.1.0 (November 22, 2023)

xxeserver based on XMLmind XML Editor v10.6.

Enhancements:

- Typing text using a CJK *Input Method Editor* (IME) now works but has limitations and bugs. For example, it's not possible to replace the text selection simply by typing text using the IME.
- Interactive commands written in JavaScript™ which are specific to a configuration are now dynamically loaded when needed. DocBook's "**Link callouts**" is such command. This short and simple command written in JavaScript first displays a dialog box letting the user choose an ID prefix, then it invokes its server-side counterpart (the command written in Java™ used by **XXE** Desktop) to actually do the job.
- Upgraded JavaScript module `browser-fs-access` to version 0.35.
- The following content objects (CSS proprietary extensions) are now supported: `command-button` and all its variants: `command-menu`, `insert-button`, `insert-after-button`, `insert-before-button`, `insert-same-after-button`, `insert-same-before-button`, `replace-button`, `convert-button`, `wrap-button`, `delete-button`, `add-attribute-button`, `set-attribute-button`, `remove-attribute-button`.
- **xxeserver** embeds Jetty in order to implement an HTTP and WebSocket server. Upgraded Jetty to version 11.0.17.

Bug fixes:

- CSS extension property `collapsible:yes` was not honored for elements having an `inline-block` or `inline-table` display.

v1.0.0 (September 1, 2023)

xxeserver based on XMLmind XML Editor v10.5.

Bug fixes:

- After many user actions, the document view was automatically scrolled to show the location of the caret, which was generally useless and annoying.
- Replaced `<xxe-client>/@systemselection` by much simpler `<xxe-client>/@button2pastestext`. We could not get `@systemselection="native"` (e.g. with Chrome on Linux) to work satisfactorily because it seems there is no way to update the *X Window Primary Selection* without updating the System Clipboard at the same time.

v1.0.0-beta4 (August 2, 2023)

xxeserver based on XMLmind XML Editor v10.4.3 (not publicly released).

Enhancements:

- Dragging the column separator found at the right of a table cell may now be used to resize the table column containing this cell. Note that this works even when a table cell has no border, hence no visible column separator.

Bug fixes:

- When `<xxe-client>/@systemselection` was set to "`emulate`" (e.g. Firefox on Linux), dragging the mouse over some text selected just a couple of characters then the text selection stopped by itself.

v1.0.0-beta3 (July 4, 2023)

xxeserver based on XMLmind XML Editor v10.4.2 (not publicly released).

Enhancements:

- Clicking inside the image representing the view of an image element now adds resize handles around this image. Dragging a resize handle lets you interactively resize the image element (DocBook example: `<imagedata>`) without having to manually change any of its attributes (DocBook example: `@contentwidth`, `@contentdepth`).

  The aspect ratio of an image element resized this way is automatically preserved. Drag a resize handle while pressing the `Ctrl` key (`Cmd` key on the Mac) if you do not wish to preserve its aspect ratio (i.e. if you want to distort the image).

- Added an **Options** submenu to the menu of the sample XML editor application. For now, this submenu only contains a single checkbox: **Autosave**. This check box lets the user turn the autosave feature on and off at will. This checkbox is disabled (grayed) unless the autosave feature has been specified and configured using attribute `@autosave`.
- Shift-clicking on an element name displayed by the node path bar now selects all the child nodes of this element. This is a handy alternative to using keyboard shortcut `Escape ArrowDown`.
- **xxeserver** embeds Jetty in order to implement an HTTP and WebSocket server. Upgraded Jetty to version 11.0.15.

Bug fixes:

- When testing newest Safari against **XXEW**, its “peculiar” Web Socket client caused **xxeserver** to raise a `org.eclipse.jetty.io.EofException` and from time to time —randomly— this completely blocked **xxeserver**.

v1.0.0-beta2 (May 29, 2023)

`xxeserver` based on XMLmind XML Editor v10.4.1 (not publicly released).

Enhancements:

- Added an *autosave* facility to the sample XML editor application. Note that this autosave facility is *not* enabled by default. See new attribute `@autosave`.
- Added "**Show Element Reference**" to the menu of the sample XML editor application.
- Added a "**Comparison of revisions**" information item to the tooltip of the document icon of the node path bar (if this feature has been enabled for this document being edited).
- Firefox: slightly improved the clipboard integration. **XXEW** now updates the system clipboard when needed to but, unlike Chrome, still cannot read its contents.
- The "**Paste from Word Processor or Browser**" add-on is now supported by XMLmind XML Editor Web Edition (**XXEW**) and is included in all **XXEW** distributions. As a consequence, a new "**Paste from Word Processor or Browser**" menu item has been added to the ◢ menu found at the bottom/right of the DocBook, DITA Topic and XHTML toolbars.

Restriction

Please note that this add-on, when used by **XXEW**, is *less good at importing data copied by **MS-Word** to the clipboard* than when used by the desktop application. The add-on is strictly identical in both contexts and in theory, this should not happen. However browsers tend to discard important style information before making copied data available to web applications such as **XXEW**. For example, lists and language information are not imported as accurately as they should be.

Bug fixes:

- The dialog box displayed by command "**Command History**" did not work correctly when one of the repeatable commands had a parameter containing a newline character (example: "`textSearchReplace a[i]foo\nbar`").
- Firefox: pressing `Ctrl-SPACE` to insert a non-breaking space character (` ` or `U+00A0`) also inserted a space character.
- In some cases, the width of "`display:marker`" content generated before an element was not computed accurately enough.

v1.0.0-beta1 (May 1, 2023)

First public release. **`xxeserver`** based on XMLmind XML Editor v10.4.0.

Index