Readium SDK Content Module Interface

Jim Dovey, Digital Content Formats Evangelist
Kobo Inc.
Toronto, ON, Canada
<jdovey@kobo.com>

November 26, 2013

Contents

1 Preamble

This document introduces an interface through which the Readium SDK [SDK] attaches to an optional Digital Rights Management (DRM) module, implemented in C++. All code samples are written using the latest version of the C++ standard, [C++11], as is the SDK itself. For information on compiler support for this standard, please refer to Appendix A.

It is intended that the content of this document would enable any third-party software engineer with a licensed copy of the Readium SDK source code would be able to implement any DRM scheme in a simple fashion, and could potentially make that available in source or binary form to other licensees of the Readium SDK.

1.1 Editorial and Conformance Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119]:

> "They MUST only be used where it is actually required for interoperation or to limit behaviour which has potential for causing harm (e.g., limiting retransmissions)"

We use these capitalized terms to indicate options and requirements which would affect the interoperability of conforming Content Modules with current and future versions of the Readium SDK. These terms will not be used when describing optional parts of e.g. C++ classes, but will call out items of important consideration to implementors. For example, it is optional whether a Content Module interacts directly with operating system security resources, but its compliance with the described class structure is REQUIRED.

All elements of this specification are considered **Normative** unless specifically marked || **This section is informative.** . All normative elements are **Mandatory** to implement, except where such an element is specifically marked OPTIONAL. Finally, where **Normative** elements are described as OPTIONAL, they MAY be omitted from an implementation, but when implemented, they MUST be implemented as described.

1.2 Terms and Definitions

Content Key A symmetric key that encrypts and decrypts the content.

License An object used to govern DRM-protected content, including usage rules applied to the content and the related Content Key.

Token File A file which, instead of containing the actual publications, contains only information used to obtain a license and/or the real content file. This concept can be extended to any file format not directly supported by the Readium SDK, but from which Electronic Publication (EPUB) content can be provided.

2 Overview

This section is informative.

2.1 About the Readium Content Module Interface

The Content Module interface for the Readium SDK was designed with the primary goal of enabling alternative content formats to be integrated with the SDK, and thus with any EPUB reading system based upon that SDK. This requires two-way communication between the implementation of a content format and the rendered (or interactive) representation of that content.

The design was originally proposed in order to support DRM implementations such as Adobe's Adept and Sony's Marlin. Both of these require the ability to 'vet' content prior to loading it, to perform decryption and/or digital signature verification, and to observe and influence the interactions with the content in some way. It was determined that such an interface could be made general enough to support handling of any content, not just encrypted enclosures and license-based *token files*.

As a result, the Content Module interface provides a means by which new container types or document representations can be implemented and tested. For instance, a Content Module could be written to support EPUB publications stored within a XAR rather than ZIP containers, or it could be written to implement storage of EPUB metadata and Content Documents in Binary XML format, converting the documents to textual XHTML when loaded.

Though we anticipate the initial use of this interface will be for the implementation of DRM handling for Readium SDK-based reading systems, it is our sincere hope that it will be of considerable assistance to future research into new representation formats for the EPUB ecosystem.

2.2 A note about `std::future`

Many of the APIs described in this document return their results asynchronously by utilizing the `std::future` class from C++11. Information about this class can be found at the excellent C++ reference website [ShPtr], while details of the *future-promise paradigm* can be found on Wikipedia [FutProm].

It is the intent of this author to provide an implementation of `std::future` which allows for the use of *continuations* to asynchronously handle the value without polling. At present, the API provided by the C++11 implementation provides for a blocking fetch of the future's value or a wait-with-timeout method, but nothing to run a lambda function automatically upon the future's value being set. My intent is to create something similar to Microsoft's C++ Parallel Patterns Library, adding a `then()` method to a future object which would be invoked when the future's value is set, or when an exception occurs.

The scope and details of the new implementation are beyond the scope of this document, but it may be useful for DRMModule designers to be aware of this intended future[1] functionality.

As a guideline, this has been proposed by Microsoft as an addition to the C++17 standard in [CXX17-FUT].

[1] Pun most definitely intended, because I'm a comical fella.

2.3 Content Loading

A DRM module needs to provide certain facilities to the host SDK, and likewise needs to consume data from the SDK itself. There are a few discrete points in the content handling process at which a DRM module would want to be consulted. Before we can inspect those, however, we must first describe the content-loading process employed by the host SDK:

1. A user performs an action which references some content: display, copy, print, &c.
2. A Content Fragment Identifier (CFI) or similar reference is generated which identifies the content upon which the action is to take place.
3. The host SDK locates the content item (or subdata range within an item) represented by that reference.
4. If the data needs to be loaded from the container:
   (a) A filter chain is created, consisting of an ordered list of content-modification filters which will be used to modify the source data (i.e. the bytes stored within the Open Container Format (OCF) container) such that it becomes suitable output data for the renderer.
   (b) The item's data is passed through each filter in the chain, allowing them to perform any necessary modifications to the data in turn.
   (c) The final filtered data is obtained from the filter chain's output.
5. Once the content data has been obtained, the relevant action is performed.

With that in place, we can see where a DRM system might want to insert itself into the data flow. For example:

1. A user performs an action which references some content: display, copy, print, &c.
2. A CFI or similar reference is generated which identifies the content upon which the action is to take place.
3. The host SDK locates the content item (or subdata range within an item) represented by that reference.
   (a) The DRM module inspects the action, CFI and item to determine whether they meets certain access criteria, or whether they constitute a prohibited action.
   (b) If criteria exist and are not met, then the DRM module denies/cancels the operation.
4. If the data needs to be loaded from the container:
   (a) A filter chain is created, consisting of an ordered list of content-modification filters which will be used to modify the source data (i.e. the bytes stored within the OCF container) such that it becomes suitable output data for the renderer.
      i. The DRM module provides a decryption filter to the filter chain, and at this point it ensures it is properly prepared with the requisite parameters.

   (b) The item's data is passed through each filter in the chain, allowing them to perform any necessary modifications to the data in turn.
   (c) The final filtered data is obtained from the filter chain's output.
5. Once the content data has been obtained, the relevant action is performed.

Note that it is expected that DRM implementations will enforce rule-based restrictions based on metadata rather than actual content data. This is for performance reasons, since the metadata can be obtained easily, while the content may need to pass through several complex filters before it can be inspected. The host SDK makes the assumption that any DRM module which does not cancel the process prior to the creation of a filter chain to load new data does not intend to prohibit access to the relevant content. The DRM implementation MAY modify data through the use of a late-stage `ContentFilter`, however.

Fur the curious, there is a description of the content filter subsystem in Appendix B.

2.4 Token Files

The content-loading system outlined here doesn't cover all bases, however: some DRM formats, such as that utilized by Adobe Content Server 4, deliver books as special 'token' files, which provide information on how to authenticate and obtain the real EPUB file from another location.

To correctly handle this, it is necessary to allow a DRM module the ability to register a file-type handler. The Readium SDK actually already supports this mechanism already for archive files, so it will be a simple matter to define a type-sniffer method that a DRM module can implement to assume ownership of its own special file types. This method can utilize the C++ `std::promise` and `std::future` [ShPtr] classes to make that data available in an asynchronous manner.

The best way to encapsulate such items, I believe, would be for the DRM module to undertake the instantiation of an `ePub3::Container` or a custom subclass thereof. This has a number of benefits:

- All container handling—that is, all *file* handling—can be routed through a single asynchronous *file_path* → *Container* process within the host SDK.
- By assuming the burden of instantiating the container, the DRM module is able to explicitly load all rule sets *before* the host SDK has any opportunity to act upon the container's content.
- The DRM module can assume all ownership of any downloaded content files; the host SDK will have no access to the real EPUB file except through the DRM module and the token file.
- The DRM module has the option of storing a downloaded EPUB file somewhere and opening/returning its `Container` object immediately on subsequent requests.
- A particularly complex DRM implementation can choose to implement subclasses of all the primary EPUB object classes from the host SDK, including Container, Package, SpineItem, and ManifestItem, and even the navigation classes. This way it can provide for very fine-grained control over the capabilities and rights vended with the content itself.

In the next section we will enumerate the components of the Readium SDK's DRM API and describe how it can be utilized to provide the features outlined above.

3 Content Module API

This section defines the interface for the `ContentModule` abstract class. Each DRM system providing integration with the Readium SDK SHALL implement this interface to make its facilities available to the rest of the system, and it MUST use the interface exported by the `ContentModuleManager` class, defined in Section 3.6, to register itself with the SDK.

The Content Module API is built around an *observation* paradigm, meaning that most of its functions take place by observing certain actions of the host SDK— specifically, those concerned with the loading and subsequent presentation of content. These actions are represented by instances of the `UserAction` class, which encapsulates both the type of action and a reference to the relevant content in CFI format.

Note that all APIs described here exist within the `ePub3` namespace.

3.1 Enum Class `ActionType`

```
enum class ActionType
{
    Display                     = 0,
    Print                       = 1,
    Copy                        = 2,
    Quote                       = 3,
    Share                       = 4,
    Highlight                   = 5,
    BeginMediaOverlayPlayback   = 6,
    BeginSpeechSynthesis        = 7,
    BeginScreenReaderSynthesis  = 8,
    BeginAudioMediaPlayback     = 9,
    BeginVideoMediaPlayback     = 10,
    DisplayMediaFullscreen      = 11,
    BeginAnimation              = 12
};
```

We use an enumeration to define the types of user action known to the host SDK. This enumeration is

subject to change in the future, but the values of existing members are guaranteed to remain constant.

3.1.1 Constants

Constants

Name	Value	Description
`Display`	`0`	Display the given item to the user. May be thought of in terms of a *read right*.
`Print`	`1`	Send the given item or range of content to a printer to produce hard-copy.
`Copy`	`2`	Copy the subrange to a device's clipboard or pasteboard, from which it can be pasted as input to another application.
`Quote`	`3`	Copy the item or subrange with the intent of sending it to a social media service such as http://twitter.com or http://facebook.com. This MAY imply grant of the `Copy` action.
`Share`	`4`	Share the content with another user, in the context of lending a book or some sub-chapter. The implication of this action is that the shared item is not copied but *moved* for a period of time.
`Highlight`	`5`	Add a highlight to a given item, which can then be located later through a list of highlights or similar. This action often implies sharing, but that is not always the case.
`BeginMediaOverlayPlayback`	`6`	Initiate playback of audio content defined by the ePub file through a media overlay [EPUB3-MO].
`BeginSpeechSynthesis`	`7`	Initiate playback of computer-generated speech synthesis of a Content Document. This is kept distinct from the Media Overlay system, and is intended to allow vendors of text-only and audio-inclusive eBooks to close the *speech synthesis loophole*, requiring users to purchase the audio-inclusive edition to get audio playback.
`BeginScreenReaderSynthesis`	`8`	Begin synthesis of read-along audio for the purposes of a screen reader. This event is provided for **introspection only** — you cannot use this to disable a visually-impaired reader's screen reading software.
`BeginAudioMediaPlayback`	`9`	Begin playback of inline audio elements within a Content Document. Once playback is allowed, all other operations on the media are assumed to be implicitly permitted.
`BeginVideoMediaPlayback`	`10`	Begin playback of inline video media elements within a Content Document. Once playback is allowed, all other operations on the media are assumed to be implicitly permitted.
`DisplayMediaFullscreen`	`11`	Whether media can be displayed in full-screen rendition. This action may be posted and thus authorized in advance; for example a content filter might fire this action and alter the markup of a Content Document accordingly.
`BeginAnimation`	`12`	Begin an animation sequence defined using non-media formats such as JavaScript or SVG.

3.2 Type async_result

```
template <typename _Tp>
using async_result = std::future<_Tp>;
```

This type is provided as a convenience to implementors of Content Modules. The underlying type, at present, is `std::future`, and the type will always follow the same API as the C++11 implementation of that class. While waiting for the C++17 standard to be implemented, however, a future revision of the host SDK will include its own implementation of both `std::future` and `std::promise` which will allow for *continuations* to be applied to the future, allowing truly asynchronous processing.

The type definition is here to allow your code to adopt either underlying type dependant on the host SDK revision against which it is built.

Refer to Section 2.2 for more discussion of this topic.

3.3 Type promised_result

```
template <typename _Tp>
using promised_result = std::promise<_Tp>;
```

This type is provided as a convenience to implementors of Content Modules. The underlying type, at present, is `std::promise`, and the type will always follow the same API as the C++11 implementation of that class. While waiting for the C++17 standard to be implemented, however, a future revision of the host SDK will include its own implementation of both `std::future` and `std::promise` which will allow for *continuations* to be applied to the future, allowing truly asynchronous processing.

The type definition is here to allow your code to adopt either underlying type dependant on the host SDK revision against which it is built.

Refer to Section 2.2 for more discussion of this topic.

3.4 Class UserAction

```
class UserAction
{
public:
    UserAction(ConstManifestItemPtr item,
               const CFI& cfi,
               ActionType type=ActionType::Display)
        : m_item(item), m_cfi(cfi), m_type(type) {}
    virtual ~UserAction() {}

    virtual
    ActionType          Type()        final noexcept { return m_type; }
    virtual
    const CFI&          CFI()         final noexcept { return m_cfi; }
```

```
    virtual
    ConstManifestItemPtr ManifestItem() final noexcept { return m_item; }
    virtual
    const IRI&           IRI()          final noexcept { return m_iri; }

private:
    ActionType              m_type;
    CFI                     m_cfi;
    IRI                     m_iri;
    ConstManifestItemPtr    m_item;
};
```

3.4.1 Class Overview

The `UserAction` class encapsulates the three components of an action:

- The *type* of action being undertaken.
- The *manifest item* upon which the action will take place.
- The *CFI* referencing the content to which the action will apply.

Note that these properties are `private`, meaning that they cannot be accessed except through their accessors, and that their accessor methods are both *read-only* and *final*. This last is a new feature in C++11: any `virtual` function may be marked with the `final` keyword, meaning that the compiler will *not* allow it to be overridden by subclasses. In this way, subclasses cannot (easily) be implemented which return dummy values for these parameters.

3.4.2 `ePub3::ActionType UserAction::Type()`

Parameters

Name	Type	Description
None		

Result

Returns the type of action represented by this object.

3.4.3 `const ePub3::CFI* UserAction::CFI()`

Parameters

Name	Type	Description
None		

Result

Returns the CFI referencing the content upon which the action will take place.

3.4.4 ePub3::ConstManifestItemPtr UserAction::ManifestItem()

Parameters

Name	Type	Description
None		

Result

Returns a shared pointer to the `ManifestItem` representing the content upon which the action will take place. Note that the CFI may indicate that only a sub-range of the item's content will be used.

3.5 Class ContentModule

```
class ContentModule
    : public std::enable_shared_from_this<ContentModule>
{
public:
    ContentModule()                                 = default;
    ContentModule(ContentModule&&)                  = default;
    virtual ~ContentModule()                        = default;

    virtual
    ContentModule& operator=(ContentModule&&)       = default;

    ContentModule(const ContentModule&)             = delete;
    ContentModule& operator=(const ContentModule&)  = delete;

    ////////////////////////////////////////////
    // Token files

    virtual
    async_result<ContainerPtr>
    ProcessFile(const string& path,
                std::launch policy=std::launch::any)        = 0;

    ////////////////////////////////////////////
    // Content Filters

    virtual
    void
    RegisterContentFilters()                                = 0;

    ////////////////////////////////////////////
    // User actions

    virtual
    async_result<bool>
    ApproveUserAction(const UserAction& action)             = 0;
```

```
};
```

3.5.1 Class Overview

The `ContentModule` class is an abstract class that DRM implementors MUST subclass to hook into the host SDK. This class is the primary means by which the host SDK interacts with the high-level facilities of your DRM system.

The Content Module subsystem of the SDK provides a general-purpose facility for the creation of handlers for different file types. For example, if a custom document solution used the EPUB layout within a XAR container rather than a ZIP container, then a Content Module would provide the integration of that format into the host SDK. For DRM implementors, it simply provides a useful integration point, since it allows for the handling of non-EPUB file types such as the *token files* used by DRM schemes such as Adobe's Adept.

3.5.2 `async_result<ContainerPtr> ContentModule::ProcessFile(const ePub3::string& path)`

This method takes the filesystem location of a file to be processed and will instantly return an instance of `async_result`. The actual processing of the file will take place asynchronously, in a manner unspecified by this document (though note the `ePub3::executor` class and its subclass `ePub3::thread_pool`).

The implementation would be able to make use of a number of C++11 or C++17 standard library facilities, most revolving around the `std::promise` class or the host SDK's own implementation of the proposed C++17 implementation [CXX17-FUT]. Both `std::async` and `std::packaged_task` or their host SDK-provided counterparts will provide this functionality for you, vending a `std::future` instance for you to return.

For more information, you can refer to the online documentation on `std::future` [ShPtr]. Additionally, please read the information in Section 2.2 for some important details on some intended updates to the future/promise API.

The asynchronous implementation would inspect the file at the provided path to determine if it is a token file understood by the current DRM scheme. This document places no restriction upon the heuristics used here, or on the document types handled. For example, an implementation MAY choose to handle pure `.epub` files themselves, using nothing more than the standard XML encryption and digital signature [XML-ENC, XML-DSig] items within the EPUB container [**?**]. Further, an implementation MAY perform any other filesystem or network access that it deems necessary including, but not limited to, downloading new files, extracting the contents of archives, or searching for other files based on the input Fully-Qualified Path Name (FQPN).

An example implementation can be seen in Section C.4 of the appendices.

Parameters

Name	Type	Description
path	`const ePub3::string&`	A FQPN giving the location of the file to process.

Result

This returns a new `std::future` object which will vend either `nullptr` or a valid `ContainerPtr` at some future time.

3.5.3 void ContentModule::RegisterContentFilters()

This method is called when the host SDK loads a Content Module, so that the module can register its internal ContentFilter subclasses for use in the content loading system.

While modules MUST provide an implementation of this method, they MAY choose to define it as an unsupported operation should they have no need for a discrete content filter (for instance, if they subclass ManifestItem and perform decryption directly there). Details on defining an unsupported operation can be found in Section 3.5.5.

An example implementation can be seen in Section C.5 of the appendices.

Parameters

Name	Type	Description
None		

Result

None.

3.5.4 async_result<bool> ApproveUserAction(const ePub3::UserAction&)

This method is the primary source of rule enforcement for a DRM provider. All user actions will be sent to the ApproveUserAction() method of the appropriate DRM provider for asynchronous processing.

The method is expected to operate asynchronously, under the assumption that some form of processing must take place to determine whether the action can be authorized. In a simple case (e.g. all copying is disallowed), the implementation MAY create a async_result instance directly, set its value, and return it. The host SDK's own implementation of C++17's std::future will provide an implementation of make_ready_future to perform this operation without the necessity of creating a promised_result instance.

Normally, however, the host SDK will expect to be given an invalid async_result object (i.e. one with no result yet set) and it will not perform any further processing of the action until the result is provided.

Please see Section 2.2 for some important information about futures and promises in the Readium SDK.

Parameters

Name	Type	Description
action	const ePub3::UserAction&	An object describing an action to take place and the content upon which it will operate.

Result

Returns (asynchronously via async_result) true if the action is allowed, or false if it is prohibited. In the event of a prohibition, the module can call ContentModuleManager::DisplayMessage() (cf. Section 3.6.4) to present information to the user concerning the restriction.

3.5.5 Unsupported Operations

If in any case a particular API is not required or supported for a particular Content Module, then that module SHALL throw a `std::system_error` instance, with a value of `std::errc::operation_not_supported`. An example of this can be seen in Listing 1. The Readium SDK asserts that it will correctly handle any such exception, though the details of that handling is not defined here, and is subject to change without notice.

```
using namespace std;
throw system_error(make_error_code(errc::operation_not_supported), "...");
```

Listing 1: Indicating an unsupported DRM operation

3.6 Class ContentModuleManager

```
class ContentModuleManager
{
public:
    ContentModuleManager();
    virtual ~ContentModuleManager();

    ContentModuleManager(const ContentModuleManager&)               = delete;
    ContentModuleManager(ContentModuleManager&&)                    = delete;
    ContentModuleManager& operator=(const ContentModuleManager&)    = delete;
    ContentModuleManager& operator=(ContentModuleManager&&)         = delete;

    ////////////////////////////////////////////////
    // Accessing the singleton instance
    static
    ContentModuleManager*
    Instance() noexcept;

    ////////////////////////////////////////////////
    // Registering content module implementations

    void
    RegisterContentModule(std::shared_ptr<ContentModule> module,
                          const string& name) noexcept;

    ////////////////////////////////////////////////
    // Services for DRM implementations

    static
    void
    DisplayMessage(const string& title,
                   const string& message) noexcept;

    static
```

```
    async_result<Credentials>
    RequestCredentialInput(const CredentialRequest& request);

private:
    static std::unique_ptr<ContentModuleManager>    s_instance;
};
```

3.6.1 Class Overview

The ContentModuleManager class is a singleton which handles the registration and management of any and all ContentModule subclasses. The interface shown here is only partial; there are other APIs available only to the host SDK itself.

3.6.2 static ContentModuleManager* Instance()

This method provides access to the singleton instance of the ContentModuleManager class. You will need to use this method to fetch that instance in order to register your ContentModule object.

Parameters

Name	Type	Description
None		

Result

A raw pointer to the global unique ContentModuleManager instance.

3.6.3 void RegisterContentModule(std::shared_ptr<ContentModule> module, const string& name)

You will need to call this method exactly once to register a single instance of your ContentModule subclass. Note that the pointer you pass in is actually an instance of std::shared_ptr; the ContentModule base class subclasses std::enable_shared_from_this to enable easy access to a shared pointer by calling shared_from_this() on your subclass. Note that this functions only once your instance has been introduced to the C++11 shared pointer subsystem– the easiest way is to replace `new MyObject(args...)` with `std::make_shared<MyObject>(args...)`.

The example module implementation includes an example of the correct use of this method in Section C.3 of the appendices.

Parameters

Name	Type	Description
module	std::shared_ptr<ContentModule>	A reference-counted pointer wrapping a heap-allocated instance of a ContentModule subclass.
name	const string&	A name for your Content Module. This is used to aid in debugging and in identifying your module. Multiple calls to RegisterContentModule() with the same name parameter will replace the existing module.

Result

None.

3.6.4 static void DisplayMessage(const string& title, const string& message)

Content Modules can present messages to the user by calling this method. It is expected that the reading system implementation will ultimately display the message in a dialog or popup appropriate to the host operating system. The Host SDK will act as an intermediary here, though the exact implementation is outside the scope of this document.

Parameters

Name	Type	Description
title	`const ePub3::string&`	The title for the message dialog. This should be a short, clear phrase of at most a few words.
message	`const ePub3::string&`	A descriptive message. Note that on some devices there may not be a lot of room to display longer messages, so it is advisable to be brief here.

Result

None.

3.6.5 static async_result<Credentials> RequestCredentialInput(const CredentialRequest& request)

When a Content Module requires some form of user input, either as a username/password pair, or as some other form of identity confirmation, then it uses this method to prompt the user. The host SDK will make the details of the credential available to the reading system implementation such that it can be presented in a manner appropriate to the host OS.

Due to the potentially complex nature of credentials and the open-ended design of this API, the credential request is represented by a `CredentialRequest` object. This class is described in more detail in Section 3.7.

The example module implementation includes an example of the correct use of this method in Section C.6 of the appendices.

Parameters

Name	Type	Description
request	`const ePub3::CredentialRequest&`	An object codifying the credentials being requested along with some identification and presentation metadata.

Result

The method returns a `Credentials` object asynchronously via an `async_result` object. The value will be available to read from the future once the user has either provided the requested input or has cancelled the dialog.

Please see Section 2.2 for some important information about futures and promises in the Readium SDK.

3.7 Class CredentialRequest

```
class CredentialRequest
{
public:
    enum class Type : uint8_t
    {
        Message,
        TextInput,
        MaskedInput,
        Button
    };

    using ButtonHandler =
        std::function<void(const CredentialRequest*, size_t)>;

public:
    CredentialRequest(const string& title,
                      const string& message);
    CredentialRequest(const CredentialRequest& o);
    CredentialRequest(CredentialRequest&& o);
    virtual ~CredentialRequest();

    CredentialRequest& operator=(const CredentialRequest&);
    CredentialRequest& operator=(CredentialRequest&&);

    const string&
    GetTitle() const noexcept;

    const string&
    GetMessage() const noexcept;

    size_t
    AddCredential(const string& title,
                  bool secret = false,
                  const string& defaultValue = string::Empty);
    size_t
    AddButton(const string& title, ButtonHandler handler);

    Type
    GetItemType(size_t idx) const;

    const string&
    GetItemTitle(size_t idx) const;

    const Crendentials&
    GetCredentials() const noexcept;

    void
```

```
    SignalCompletion() noexcept;

    void
    SignalException(std::exception& exc) noexcept;

private:
    class Component
    {
    public:
        Component(Type type, const string& title);
        Component(Type type, string&& title);
        Component(const Component& o);
        ~Component() {}

        void SetSecret(bool isSecret);
        bool GetSecret() const noexcept;

        void SetDefaultValue(const string& defaultValue);
        void SetDefaultValue(string&& defaultValue);
        const string& GetDefaultValue() const noexcept;

    private:
        Type        m_type;
        string      m_title;
        bool        m_secret;
        string      m_default;

        friend class CredentialRequest;

    };

    std::vector<Component>          m_components;
    Credentials                     m_credentials;
    promised_result<Credentials>    m_promise;
};
```

3.7.1 Class Overview

The `CredentialRequest` class encapsulates the information and metadata required to prompt for user input of credential-related data. It defines a means of describing a prompt dialog (or panel, pane, screen, sheet, &c.) including messages, text input fields (both clear and secure), and buttons.

Its API allows a Content Module implementation to provide a title and message for the prompt, and to add credential input fields and buttons. The buttons are handed synchronously through the invocation of a delegate function supplied by the caller, which receives the content of the input fields as a parameter.

3.7.2 Enum Class Type

When introspecting the contents of a `CredentialRequest`, the type of an item is important: is it a button, the prompt's title, or an input field? This enumeration defines the available types, as returned from the

GetItemType() method defined in Section 3.7.9.

Title The title of the presented dialog. There can only be one of these items, and it is set only through the CredentialRequest constructor (cf. 3.7.4).

Message The message presented to the user. There can be only one of these items, and it is set only through the CredentialRequest constructor (cf. 3.7.4).

TextInput A textual input field. This field is not masked in any way, so the input text is visible as the user types it. These are added using the AddCredential() method (cf. 3.7.7).

MaskedInput A textual input field which hides its input. Commonly used for password input. Added using the AddCredential() method (cf. 3.7.7).

Button A button with a title and a delegate callback function associated with it. Added using the AddButton() method (cf. 3.7.8).

3.7.3 Delegate Function Type ButtonHandler

This type defines a delegate function to be called when a button on the credential prompt is pressed. It takes as arguments a raw pointer to the button's parent CredentialRequest and the index of the button that was pressed. Note that this index is relative to all components of the prompt– it is **not** a button index.

You can obtain the title of the button pressed by calling GetItemTitle() on the supplied CredentialRequest object, defined in Section 3.7.10.

3.7.4 CredentialRequest(const string&, const string&)

Constructs a new credential request with a title and informative message.

The example module implementation includes an example of the correct use of this method in Section C.6 of the appendices.

Parameters

Name	Type	Description
title	const ePub3::string&	The title of the credential prompt.
message	const ePub3::string&	An informative message for the user. This should describe succinctly the details requested and the reason for the request.

Result

A new CredentialRequest object.

3.7.5 const ePub3::string& GetTitle()

Retrieves the title of the credential request prompt.

Parameters

Name	Type	Description
None		

Result

The displayed in the credential request prompt, as presented to the user.

3.7.6 const ePub3::string& GetMessage()

Retrieves the message from the credential request prompt.

Parameters

Name	Type	Description
None		

Result

The message from the credential request prompt, as presented to the user.

3.7.7 size_t AddCredential(const ePub3::string&, bool, const ePub3::string&)

Adds a credential input field to the request object. The credential field requires a title, and may have a default value. Additionally, the field may have visible or masked input.

The example module implementation includes an example of the correct use of this method in Section C.6.1 of the appendices.

Parameters

Name	Type	Description
title	`const ePub3::string&`	The title (name) for the credential input field.
secret	`bool`	Pass `true` to make this input field use masked input, so the typed text is not visible on screen. The default is `false`.
defaultValue	`const ePub3::string&`	An optional default value. This could be used to pre-populate known data values (such as a username) or as a placeholder such as "Enter your username or email address".

Result

Returns the index of the added credential. This can be passed into the `GetItemType()` (see Section 3.7.9) and similar methods.

3.7.8 size_t AddButton(const ePub3::string&, ButtonHandler

Adds a button to the credential request, with a title and a delegate handler function.

The example module implementation includes an example of the correct use of this method in Section C.6.2 of the appendices.

Parameters

Name	Type	Description
title	const ePub3::string&	The title of the button.
handler	ButtonHandler	A std::function object wrapping a function, functor, or lambda which will be called when the button is clicked/tapped.

Result

Returns the index of the added button. This can be passed into the GetItemType() (see Section 3.7.9) and similar methods.

3.7.9 Type GetItemType(size_t)

Retrieves the type of the item at the given index. If the index is out-of-bounds then a std::out_of_bounds exception is thrown.

Parameters

Name	Type	Description
idx	size_t	The index of the item to inspect.

Result

The type of the item at the specified index, as defined in Section 3.7.2.

3.7.10 const ePub3::string& GetItemTitle()

Retrieves the title of the item at the given index. If the index is out-of-bounds then a std::out_of_bounds exception is thrown.

Parameters

Name	Type	Description
idx	size_t	The index of the item to inspect.

Result

The title of the item at the specified index.

3.8 Class Credentials

```
typedef std::map<string, string>        Credentials;
typedef Credentials::value_type         CredentialType;
```

3.8.1 Type `Credentials`

DRM credentials are represented simply as a map of key-value pairs. Both keys and values are strings, though the `CredentialType` (cf. 3.8.2) is provided as a type to use in client code which will remain available should credentials be recorded as non-string types in a future revision.

3.8.2 Type `CredentialType`

This type gives a constant name to the type of a credential. At present this is an `ePub3::string` object, though this may change in future based on input from other Readium contributors.

A C++11 Compiler Support

The Readium SDK is implemented using the features of the most recent revision of the C++ standard [C++11]. At the time of writing, most compilers have quite good support for this standard across either the compiler or standard library layers. This section will enumerate the major differences in support across the major compilers: GCC, Microsoft Visual C++, and LLVM Clang.

The Readium SDK provides a number of preprocessor macros that you can use to test for certain features and capabilities, as well as some for the availability of certain library features such as the C++ `std::locale` and `std::regex` classes and the implementation of the `emplace` function in standard containers.

In the case of compiler features, you will use the `EPUB_COMPILER_SUPPORTS()` macro, and for library features you can use the `EPUB_HAS()` macro. Each of these takes a single parameter identifying the capability to test; the former uses values from Table 1, the latter from Table 2.

The tables below enumerate the features of the C++11, C++14 (proposed), and C++17 (proposed) language and standard library that are used by the Readium SDK, along with the macros that you can use to test for them, followed by the versions of each of the three major compilers that support them. The first table lists compiler features, and the second lists standard library features.

Note Where a language or library feature is not enumerated in the tables below, we have assumed that it will always be available. For instance, we assume that all compilers being used with the Readium SDK will support rvalue references and rvalue-based initialization and assignment.

C++11 Language Features				
Description	**Macro**	**GCC**	**Clang**	**MSVC**
Non-static data member initializers	`CXX_NONSTATIC_MEMBER_INIT`	4.7	3.0	VS2013
Variadic Templates	`CXX_VARIADIC_TEMPLATES`	4.3	2.9	VS2013
Initializer lists	`CXX_INITIALIZER_LISTS`	4.4	3.1	VS2013
Static assertions	`CXX_STATIC_ASSERT`	4.3	2.9	VS2010
Default arguments for function templates	`CXX_DEFAULT_TEMPLATE_ARGS`	4.3	2.9	VS2013
Alias templates	`CXX_ALIAS_TEMPLATES`	4.7	3.0	VS2013
Generalized constant expressions	`CXX_CONSTEXPR`	4.6	3.1	-
Delegating constructors	`CXX_DELEGATING_CONSTRUCTORS`	4.7	3.0	VS2013
Explicit conversion operators	`CXX_EXPLICIT_CONVERSIONS`	4.5	3.0	VS2013
Unicode string literals	`CXX_UNICODE_LITERALS`	4.5	3.0	-
Raw string literals	`CXX_RAW_STRING_LITERALS`	4.5	3.0	VS2013
User-defined literals	`CXX_USER_LITERALS`	4.7	3.1	-
Defaulted functions	`CXX_DEFAULTED_FUNCTIONS`	4.4	3.0	VS2013
Deleted functions	`CXX_DELETED_FUNCTIONS`	4.4	2.9	VS2013
Local and unnamed types as template arguments	`CXX_LOCAL_UNNAMED_TEMPLATE_ARGS`	4.5	2.9	VS2010
Explicit virtual overrides	`CXX_OVERRIDE_CONTROL`	4.7	3.0	VS2012
Thread-local storage	`CXX_THREAD_LOCAL`	4.8	3.3	-

Table 1: C++ Language Feature Support

C++11 Standard Library Features				
Description	**Macro**	**GCC**	**Clang**	**MSVC**
Regular expressions	Use `REGEX_INCLUDE` and `REGEX_NS`	4.8.2	3.0	VS2012
Locales and encoding conversions	Use `LOCALE_INCLUDE` and `LOCALE_NS`	-	3.0	VS2012
`std::map::emplace()`	`CXX_MAP_EMPLACE`	4.8	3.0	VS2012
C++14 Standard Library Features				
Description	**Macro**	**GCC**	**Clang**	**MSVC**
Compile-time integer sequences	Provided by Readium[2]	4.8.2?	3.4	-
`std::make_unique()`	Provided by Readium[2]	4.8.2?	3.4	VS2013
C++17 Standard Library Features				
Description	**Macro**	**GCC**	**Clang**	**MSVC**
A utility class for optional objects	Provided by Readium[2]	-	-	-
`std::executor` class	Provided by Readium[2]	-	-	-
Improvements to `std::future`	Provided by Readium[2]	-	-	-

[2]The Readium SDK provides an implementation of these classes and functions under the `ePub3` namespace.

Table 2: C++ Standard Library Feature Support

B About Readium Content Filters

The `ContentFilter` abstract superclass provided by the Readium SDK forms the base of much of the more intricate work for content modules.

B.1 ContentFilter Overview

This section is informative.

A content filter exists to serve a single purpose: it takes a stream of data as input and outputs that data, possibly with modifications. These modifications can take any form, but common examples include:

- Encryption/Decryption.
- Digital Signatures.
- Validation.
- Content Injection.

The Readium SDK ships with three built-in content filters: one implements EPUB3 Font Obfuscation [EPUB3-FO], one modifies content containing `<epub:switch>` constructs [EPUB-SW] to contain only the selected case, and the last transforms `<object>` tags into `<iframe>` tags referencing a DHTML media handler. In the future the SDK will also implement filters which handle content that is encrypted [XML-ENC] or digitally signed [XML-DSig] according to the Open Container Format specification [OCF].

It is anticipated that most content filters provided by third parties will handle encryption in one form or another.

B.2 Filter Chains

This section is informative.

Filters operate in *chains*. These are lists of filters connected together in priority order, and data is passed directly between them. At one end of the chain is the data contained in the EPUB container, and at the other is a stream object held by the Reading System implementation.

As data is loaded from the container, it is passed to the first filter in the chain– the one with the highest priority. This filter then (potentially) modifies the data, and its output then feeds into the next filter in the chain. Eventually, the output from the last filter in the chain is sent as input to the stream to be read by the Reading System.

B.2.1 Priorities

Each filter operates at a single priority, which is a value in the range [0..1000]. There are a number of predefined priority values, each of which is intended as a threshold to better enable prioritization by the author of each filter:

1000	`ContentFilter::MustAccessRawBytes`	The highest priority. Use this if your filter absolutely must see the bytes from the container unaltered.
750	`ContentFilter::EPUBDecryption`	The priority at which built-in handling obfuscated fonts [EPUB3-FO] and of XML encryption [XML-ENC] and digital signatures [XML-DSig] takes place.
500	`ContentFilter::SwitchStaticHandling`	Processing of `<epub:switch>` data happens at this priority.
250	`ContentFilter::ObjectPreprocessing`	Handling of EPUB media handlers and `<object>` tags happens here.
100	`ContentFilter::ValidationComplete`	All validation-based filters MUST have higher priorities than this value. Anything which requires that it operate on valid content should have a priority of 100 or lower.

B.2.2 Instances and Setup

A `ContentFilter` subclass will be allocated *at most* once for each `Package` that is loaded. This single instance will then be used to process all applicable data loaded through that package.

Your instance may be used to process two or more streams of resource data concurrently, for which reason we provide the `FilterContext` class (cf. Section B.3). This is a simple superclass from which you can derive for the purposes of storing per-resource data. For instance, while a Font Obfuscation [EPUB3-FO] filter uses the same obfuscation key for all content within a given package, encryption or digital signature filters might utilize separate keys for each resource. In this case, a filter would create a `FilterContext` subclass to record that information.

Each filter is allocated using a *factory function* which you provide when registering your filter subclass with the singleton `FilterManager`. The factory function is given a reference to the `Package` for which the filter is to be instantiated, and it can inspect that package, its contents, and its owning `Container` instance to determine whether your filter applies to any content that would be loaded from the package. If your filter does not apply to any resources there, your factory should return `nullptr` rather than instantiating a filter instance that will never be used.

Any allocated filters are placed into a priority queue attached to the `Package` instance itself. When resources are loaded, a discrete *filter chain* is created to process the data for that resource as it is streamed from the container to the client Reading System. Each filter attached to the package is queried, again in priority order, to determine whether it should apply to that resource; a filter provides a *type sniffer* function (cf. Section B.3.2) to make this determination. This function is passed the `ManifestItem` representing the resource being loaded, and it will return either `true` or `false` to indicate whether the filter should be used to process this data.

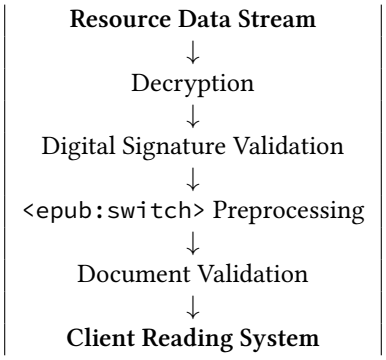
If the type sniffer returns `true`, the filter will be added to the chain used for this resource, and it will be asked to create a `FilterContext` object containing per-resource data (cf. Section B.3.4). This is not required, but if returned it should be a raw pointer allocated with the global C++ `operator new`; the filter chain will deallocate it when the resource data has all been processed using `operator delete`. If you don't need any per-resource data, then simply return `nullptr`.

Once all the filters have been assigned to the resource's chain, data is read from the container and passed in chunks through each of the filters in turn using the filters' `FilterData()` method, defined in Section B.3.8. This method is passed the `FilterContext` object that filter allocated earlier, if one was supplied. Each filter will receive the output of the filter preceding that one in the chain. Eventually, the output of the last filter in the chain is provided to the stream object held by the client Reading System, ready to be displayed or otherwise processed there.

Note Any filter in the chain can request that it receive *all* the data from a resource in a single pass (cf. Section B.3.5). This is useful, for example, when you need to scan across the entire document and need to ensure that nothing you're matching is split across the boundary between two input blocks. The `<epub:switch>` and EPUB Media Handler filters both make use of this feature, for example.

B.2.3 Example Filter Chain Flow

For example, consider a filter chain which incorporates filters for decryption, signature validation, preprocessing of `<epub:switch>` constructs, and finally document schema validation. The data will 'bubble' through the filters in a serial fashion as shown below:



B.3 Class ContentFilter

Below is the public interface for the `ContentFilter` abstract class.

```
class ContentFilter
{
public:
    typedef uint32_t FilterPriority;
```

```
    typedef std::function<bool(ConstManifestItemPtr item)> TypeSnifferFn;
    typedef std::function<ContentFilterPtr(ConstPackagePtr package)> TypeFactoryFn;

    ContentFilter() = delete;
    ContentFilter(TypeSnifferFn sniffer)
        : _sniffer(sniffer)
        {}
    virtual
    ~ContentFilter()
        {}

    ContentFilter(const ContentFilter& o)
        : _sniffer(o._sniffer)
        {}
    ContentFilter(ContentFilter&& o)
        : _sniffer(std::move(o._sniffer))
        {}


    virtual
    FilterContext* MakeFilterContext(ConstManifestItemPtr item) const
        { return nullptr; }

    virtual
    bool RequiresCompleteData() const
        { return false; }

    virtual
    TypeSnifferFn TypeSniffer() const
        { return _sniffer; }

    virtual
    void SetTypeSniffer(TypeSnifferFn fn)
        { _sniffer = fn; }

    virtual
    void * FilterData(FilterContext* context,
                      void *data, size_t len,
                      size_t *outputLen)              = 0;
};
```

B.3.1 Factory Function Type TypeFactoryFn

This type defines a delegate function to be called when a filter is to be instantiated. The factory function is given an instance of Package, and can use that to determine whether the filter will be needed to process any of that package's content.

If the filter is not required, the factory function should return `nullptr`. Otherwise, it should return create

and return a new instance of the filter using `std::make_shared()`.

B.3.2 Type-Sniffer Function Type `TypeSnifferFn`

This type defines a delegate function to be called to determine whether a given filter instance should be used to process a given resource, represented by a `ManifestItem`. The function should check the item to be loaded, and return `true` if the filter should be applied, or `false` if not.

B.3.3 `ContentFilter(TypeSnifferFn sniffer)`

A ContentFilter instance is instantiated with its (required) type-sniffer function.

The reasoning behind this is so that each instance might be given its own sniffer function, for example as a lambda capturing some variables from its local scope.

Parameters

Name	Type	Description
sniffer	`TypeSnifferFn`	The filter's type-sniffer delegate function.

B.3.4 `FilterContext* MakeFilterContext(ConstManifestItemPtr item) const`

Since content filters are instantiated once per package and used to process multiple resources—possibly in parallel—this method is provided to allow a filter subclass to create a custom object to hold its per-resource variables.

Parameters

Name	Type	Description
item	`ConstManifestItemPtr`	The `ManifestItem` to be processed by the filter.

Result

The return value from this function is a subclass of the (essentially empty) `FilterContext` class, which MUST be allocated using the global C++ `operator new`. Implementors may place whatever they like in this object, giving it as simple or as complex an interface and implementation as they choose. The caller of this function treats the returned object as a raw pointer, and will reclaim it using `operator delete` when the current resource has completed processing.

B.3.5 `bool RequiresCompleteData() const`

This method, which is defined in the base class to return `false`, may be overridden by implementors of custom content filters to return `true`. If they do so, then their `FilterData()` method will be called exactly once for each resource, with the entire data from that resource passed in memory.

Implementors SHOULD attempt to process data in small pieces rather than overriding this method and operating on the entire resource. In particular, video and audio files, as well as some image files, can take up a very large amount of memory if loaded in a single chunk. Additionally, processing an entire resource

in one pass can cause a bottleneck in the filter chain, reducing the ability for the host SDK to parallelize the processing work efficiently.

Result

Returns `true` to request that all data be provided in a single pass, `false` to use piecemeal handling.

B.3.6 `TypeSnifferFn TypeSniffer() const`

This method is used by the Readium SDK to obtain the type-sniffer function for a filter. Subclasses normally need not override it.

Result

The type-sniffer function, as passed to either the constructor or to `SetTypeSniffer()`.

B.3.7 `void SetTypeSniffer(TypeSnifferFn fn)`

This method can be used to change the type-sniffer in use by a filter. It is not used by the Readium SDK itself, and is available for implementors' exclusive use.

B.3.8 `void* FilterData(FilterContext* context, void* data, size_t len, size_t* outputLen)`

This function is where the filtering process actually takes place. A filter will receive data into this function, and it MUST process that data synchronously and return a pointer to the resulting modified data. The length of the modified data is returned by reference through the `outputLen` parameter, which implementors MUST set.

Where possible, filters SHOULD try to modify the data in-place, i.e. within the `data` parameter, rather than by allocating a new output buffer. If a new buffer must be allocated, then implementors MUST allocate it as an array of bytes using `operator new []`. The caller will detect that a new buffer was returned, and will deallocate it using `operator delete []` after casting it to `unsigned char*`.

Parameters

Name	Type	Description
context	`FilterContext*`	The context data created by this filter as returned from `MakeFilterData()`.
data	`void*`	The resource data upon which to operate.
len	`size_t`	The number of bytes in `data`.
outputLen	`size_t*`	On output, this should contain the number of bytes in the buffer returned from the method.

Result

The method should return a pointer to the filtered data. Ownership of the returned pointer passes to the caller.

C Example Content Module (DRM Implementation)

This section is informative.

A sample implementation of a Content Module is included in this section. The task undertaken here is to load a publication from a DRM license *token file*, and the process is broadly modelled after that used by clients of the Adobe Content Server.

C.1 The Client's System Interface

The sample itself has been built around a theoretical private API which implements the guts of a DRM system. That API is shown here. For the sake of illustration, the private API uses callback functions to implement asynchronous behaviour; the example code will show how to integrate such APIs into the host SDK's promise/future system.

```
#include <string>
#include <functional>

class License
{
public:
    License(void* bytes, std::size_t len);
    License(const License&);
    ~License();

    License& operator=(const License&);

    const std::string& GetContentIdentifier() const;
    const std::string& GetContentURL() const;

    static void ForgotPassword(const std::string& account,
                               std::function<void()> completion);

};

class Decryptor
{
public:
    Decryptor(const License& license);
    Decryptor(const Decryptor&);
    ~Decryptor();

    Decryptor& operator=(const Decryptor&);

    // may return data in a buffer allocated with operator new[], or in input buffer
    void* DecryptBlock(void* bytes, std::size_t len, std::size_t *decryptedLen);

};
```

```
35  class BadTokenFile
        : public std::logic_error
37  {
    public:
39      BadTokenFile(const char* s) : std::logic_error(s) {}
        BadTokenFile(const std::string& s) : std::logic_error(s) {}
41  };

43  class Token
    {
45  public:
        Token(const std::string& path);
47      ~Token();

49      bool NeedsCredentials() const;

51      const std::string& GetAccountName() const;
        bool VerifyPassword(const std::string& password,
53                          const std::string& account = std::string());
        bool VerifySignature(void* sigBytes, std::size_t sigLen);
55
        License RetrieveContentLicense() const;
57
    };
59
    class Downloader
61  {
    public:
63      typedef std::function<void(const std::string& path)>    CompletionFn;

65  public:
        Downloader(const std::string& contentURL);
67      ~Downloader();

69      bool InitiateDownload(CompletionFn completion);

71  };

73  class ContentStore
    {
75  protected:
        ContentStore();
77
    public:
79      ~ContentStore();

81      static ContentStore* Instance();

83      bool HasLocalContent(const License& license);
```

```
85      std::string PathToLocalContent(const License& license);
        std::string StoreLocalContent(const License& license, const std::string& ↩
             localPath);
87
};
```

C.2 Header File

The code below shows the header file which defines both the `ContentModule` and `ContentFilter` subclasses in use by this module. In this example, only one filter is implemented; more complex modules may use more than one, each performing discrete transformations on the data.

```
#include "DRMInterface.h"
#include <ePub3/epub3.h>
#include <ePub3/content_module.h>
#include <ePub3/filter.h>

class SampleDRMModule : public ePub3::ContentModule
{
public:
    SampleDRMModule();
    virtual ~SampleDRMModule() {}

    ////////////////////////////////////////////
    // Token files

    virtual
    ePub3::async_result<ePub3::ContainerPtr>
    ProcessFile(const ePub3::string& path,
                std::launch policy=std::launch::any);

    ////////////////////////////////////////////
    // Content Filters

    virtual
    void
    RegisterContentFilters();

    ////////////////////////////////////////////
    // User actions

    virtual
    ePub3::async_result<bool>
    ApproveUserAction(const ePub3::UserAction& action);

    ////////////////////////////////////////////
    // Internal
```

```
    // Creates and registers a single instance of this ContentModule subclass.
    static void Initialize();

    // Obtains the corresponding License file for the given Container.
    const License& LicenseForContainer(ePub3::ContainerPtr& container);

private:
    bool    mFiltersRegistered;
    std::map<ePub3::ContainerPtr, License, std::owner_less<ePub3::ContainerPtr>>  ↩
         mLicenses;

    static std::string RequestCredentials(std::string& account);

};

class SampleDRMDecryptor : public ePub3::ContentFilter
{
protected:
    static bool NeedsDecryption(ePub3::ConstManifestItemPtr item);

public:
    SampleDRMDecryptor(const License& license)
        : ePub3::ContentFilter(&NeedsDecryption), mDecryptor(license)
        {}
    virtual ~SampleDRMDecryptor() {}

    // NB: in this sample, we encrypt on a container level, so we don't need
    // per-content-document context info, and therefore don't implement
    // MakeFilterContext()

    virtual void * FilterData(ePub3::FilterContext* context, void *data, size_t len, ↩
         size_t *outputLen);

private:
    Decryptor   mDecryptor;

};
```

C.3 Module Setup Code

The following code sample shows the beginning of the implementation file. Note the use of the host SDK's own implementation of `std::future`, which conforms to the latest specification for the C++17 language revision [CXX17-FUT]. Note also the use of the `REGEX_INCLUDE` macro to make use of the host SDK's `std::regex` or `boost::regex` support as appropriate for the target platform.

```
#include "DRMModuleSample.h"
#include <ePub3/content_module_manager.h>
```

```
#include <ePub3/filter_manager.h>
#include <ePub3/container.h>
#include <ePub3/package.h>
#include <ePub3/utilities/future.h>
#include <memory>

// not all platforms have std::regex, so Readium SDK will define this to be <regex>
//  or <boost/regex.hpp> (provided) as necessary
#include REGEX_INCLUDE

using namespace ePub3;

SampleDRMModule::SampleDRMModule()
    : mFiltersRegistered(false), mLicenses()
{
}

void SampleDRMModule::Initialize()
{
    auto myModule = std::make_shared<SampleDRMModule>();
    ContentModuleManager::Instance()->RegisterContentModule(myModule, "SampleDRM");
}
```

On line 22 of the above code, note that a `std::shared_ptr` instance is created and passed to the `ContentModuleManager` instance (cf. Section 3.6.3). This *reference-counted* pointer will ensure that your object remains valid as long as the system requires it.

C.4 ProcessFile() Implementation

Below is the complete text of the sample's `ProcessFile()` implementation (cf. Section 3.5.2). This code will be broken down for discussion following the listing.

```
async_result<ContainerPtr>
SampleDRMModule::ProcessFile(const string& path, std::launch policy)
{
    Token* token = nullptr;
    try
    {
        token = new Token(path.stl_str());
    }
    catch (...)
    {
        // the constructor didn't recognize the file & threw an exception

        // return an already-setup future object containing nullptr
        // Readium's future/promise implementation includes this handy function:
        return make_ready_future<ContainerPtr>(nullptr);
    }
```

```
18      // start an async task, for example using C++11's std::async()
        return async(policy, [this, token]() -> ContainerPtr {
20          ContainerPtr result(nullptr);

22          if (token->NeedsCredentials())
            {
24              std::string account = token->GetAccountName();
                std::string password = RequestCredentials(account);
26
                if (token->VerifyPassword(password, account) == false)
28                  return nullptr;     // failed to authorise access to the content
            }
30
            License license = token->RetrieveContentLicense();
32
            std::string localPath;
34          ContentStore* store = ContentStore::Instance();
            if (store->HasLocalContent(license))
36          {
                localPath = store->PathToLocalContent(license);
38          }
            else
40          {
                Downloader downloader(license.GetContentURL());
42              std::promise<std::string> downloadPromise;
                std::future<std::string> downloadFuture = downloadPromise.get_future();
44
                // fire off the download using its built-in async API
46              downloader.InitiateDownload([&downloadPromise](const std::string& ↩
                     downloadedPath) {
                    downloadPromise.set_value(downloadedPath);
48              });

50              // already running in the background here, so block until the download
                //  is done
52              std::string downloadPath = downloadFuture.get();

54              // place the downloaded item into our local store
                localPath = store->StoreLocalContent(license, downloadPath);
56          }

58          // we have the path to the actual EPUB file now, so open it
            // we use the internal OpenContainerForContentModule() method, which
60          //  doesn't attempt to (recursively, in this case) use ContentModules to
            //  load the file
62          result = Container::OpenContainerForContentModule(localPath);
            if (!bool(result))
64              return nullptr;
```

```
66          // tell the container that we created it-- this helps our filter factory
            //  determine if it's needed
68          result->SetCreator(shared_from_this());

70          // store the container/license pair here so we can retrieve the license for
            //  our filter(s) later
72  #if EPUB_HAVE(CXX_MAP_EMPLACE)
            mLicenses.emplace(result, license);
74  #else
            // GNU libstdc++ doesn't implement emplace() on std::map as of GCC 4.8
76          mLicenses[result] = license;
    #endif
78
            // lastly, return the pointer -- std::async() takes care of the
80          //  promise/future for us
            return result;
82      });
    }
```

There's a lot going on here, though it's largely due to the fairly involved process necessary to license, obtain, and open the content based on the input token file.

C.4.1 Matching the Input File

The first task is to determine whether the input file is actually one of our token files. The `Token` class is assumed to implement this check in its constructor, throwing an exception if it is not given a valid token file:

```
Token* token = nullptr;
try
{
    token = new Token(path.stl_str());
}
catch (...)
{
    // the constructor didn't recognize the file & threw an exception

    // return an already-setup future object containing nullptr
    // Readium's future/promise implementation includes this handy function:
    return make_ready_future<ContainerPtr>(nullptr);
}
```

If the Token constructor throws an exception, the function creates an initialized `future` object holding a value of `nullptr`. This is done using the C++17 `make_ready_future()` template function, provided by the host SDK.

C.4.2 Returning an Asynchronous Result

Having decided that the file is of a type handled by this module, the module MUST return an `async_result` object (cf. Section 3.2) which will provide the loaded `Container` to the caller. It doesn't matter if the file fails to load later on– it's perfectly valid to set a value of `nullptr` through the associated `promised_result` object (cf. Section 3.3). Your module still lays its claim of ownership to the file anyway.

The simplest way to do this is to use `std::async()` (or the host SDK's C++17-based implementation), and return the result of that function. This gives you the added benefit that you need only return a `ContainerPtr` from your async function/lambda and the relevant `std::promise`/`promised_result` setup is taken care of for you.

The example here uses a lambda to include the asynchronous code inline, returning the future object like so:

```
// start an async task, for example using C++11's std::async()
return async(policy, [this, token]() -> ContainerPtr {
    ContainerPtr result(nullptr);

    «lambda body elided»

    return result;
};
```

C.4.3 Content Authorization

The next step in our DRM implementation is to obtain and verify the user's credentials and to obtain their license to access the content. The core of this process is implemented in a separate method, which you can see in Section C.6. If credentials are unavailable, the lambda returns `nullptr`, indicating that the content could not be opened.

If the credentials are already available (i.e. if the DRM system implementation already has them cached), or if they are successfully retrieved, then the license is obtained as an instance of the DRM system's `License` class:

```
if (token->NeedsCredentials())
{
    std::string account = token->GetAccountName();
    std::string password = RequestCredentials(account);

    if (token->VerifyPassword(password, account) == false)
        return nullptr;     // failed to authorise access to the content
}

License license = token->RetrieveContentLicense();
```

C.4.4 Downloading the EPUB File

Once a verified license to the content has been obtained, this DRM system needs to fetch the actual EPUB file itself. The `ContentStore` class implements the local store for any EPUB files owned by this module. This allows the module to take ownership of the publication files itself, while the host SDK and the Reading System System itself need only concern itself with the token file. In this way, the content module token file appears to *be* the content file, and the actual details of the underlying implementation are effectively hidden from the host SDK and the Reading System.

```
std::string localPath;
ContentStore* store = ContentStore::Instance();
if (store->HasLocalContent(license))
{
    localPath = store->PathToLocalContent(license);
}
else
{
    «code elided»
}
```

In this example, the `ContentStore` is first queried to determine if the EPUB file is already stored locally. If it is, then the path to that file is obtained.

If the file is not cached on the device, then the DRM system's `Downloader` class is used to fetch the file. The downloader uses a callback function to implement asynchronous behaviour, and the example above passes in a lambda function to act as the callback:

```
Downloader downloader(license.GetContentURL());
std::promise<std::string> downloadPromise;
std::future<std::string> downloadFuture = downloadPromise.get_future();

// fire off the download using its built-in async API
downloader.InitiateDownload([&downloadPromise](const std::string& downloadedPath) {
    downloadPromise.set_value(downloadedPath);
});
```

Since this method is already running in the background, we use a promise/future pair to enable the function to wait for the download to complete: the downloader callback function sets the result value onto an instance of `std::promise`, and after initiating the download this method calls `get()` on the associated future, which waits for the result to become available before returning it:

```
// already running in the background here, so block until the download
//  is done
std::string downloadPath = downloadFuture.get();
```

Once the file has been downloaded, it is placed into the local store; the assumption here is that the downloader has placed the file into a temporary location, and the store will move it to a permanent location:

```
// place the downloaded item into our local store
localPath = store->StoreLocalContent(license, downloadPath);
```

C.4.5 Returning a Container

Now that the EPUB file has been downloaded, we will open it using the host SDK's `Container` class. The host SDK provides a special factory function for the exclusive use of content modules: `OpenContainerForContentModule()`. This method, unlike the standard `OpenContainer()` or `OpenContainerAsync()`, will *not* attempt to query the installed content modules to handle the file. This avoids the potential of infinite recursion:

```
result = Container::OpenContainerForContentModule(localPath);
if (!bool(result))
    return nullptr;
```

You will note that, if the container fails to open, we immediately return `nullptr`.

Since it is useful to keep a record of which content module (if any) provided a container, the host SDK offers the `Creator()` and `SetCreator()` methods. `SetCreator()` can be called at most *once*, and content modules SHOULD set themselves as the container's creator. This allows, for instance, a `ContentFilter` subclass to inspect a container to see if it belongs to the filter's associated content module, and to then query the module for information at runtime.

The value passed to `SetCreator()` is a `std::shared_ptr` instance: the `ContentModule` base class implements the ability to obtain a valid shared pointer to itself through the `shared_from_this()` [ShFromThis] method:

```
// tell the container that we created it-- this helps our filter factory
//  determine if it's needed
result->SetCreator(shared_from_this());
```

Our module then stores the license details in a `std::map`, using the `Container` pointer as a key. The C++11 `emplace()` method provides the best way to do this, although since the GNU `libstdc++` library doesn't implement this (at least as of GCC 4.8), the host SDK provides a way to check for the availability of the method.

Once the license is stored, the `Container` pointer is returned:

```
// store the container/license pair here so we can retrieve the license for
//  our filter(s) later
#if EPUB_HAVE(CXX_MAP_EMPLACE)
mLicenses.emplace(result, license);
#else
// GNU libstdc++ doesn't implement emplace() on std::map as of GCC 4.8
mLicenses[result] = license;
#endif
```

```
// lastly, return the pointer -- std::async() takes care of the
//  promise/future for us
return result;
```

C.5 Registering Content Filters

When your content module returns an asynchronous result, its `RegisterContentFilters()` method is called. This may happen immediately after you return your `async_result`, or (if the host SDK's C++17 `std::future` implementation is being used) after it returns a valid `Container` from its asynchronous implementation.

In this example, the module keeps track of whether it has already registered its content filters through the `mFiltersRegistered` member variable. The factory function for the example's single filter is implemented here as a lambda:

```
void SampleDRMModule::RegisterContentFilters()
{
    if (mFiltersRegistered)
        return;

    FilterManager::Instance()->RegisterFilter("SampleDRMDecryptor", 1000, [this](↩
         ConstPackagePtr pkg) -> ContentFilterPtr {
        // find the container.
        ContainerPtr container = pkg->Owner();

        // fetch the ContentModule which created the container, if any.
        std::shared_ptr<ContentModule> module = container->Creator();

        // if the creator is an instance of the correct class...
        auto myModule = std::dynamic_pointer_cast<SampleDRMModule>(module);
        if (bool(myModule))
        {
            // ...query it for the license for this container and return a filter.
            return std::make_shared<SampleDRMDecryptor>(myModule->LicenseForContainer↩
                (container));
        }
        else
        {
            // ... otherwise, this package isn't ours, so we don't provide a filter ↩
                for it.
            return nullptr;
        }
    });
}
```

The lambda function receives a `Package` pointer, which it queries first for its `Container`, and then for that container's `Creator`. If the creator is an instance of the correct `ContentModule` subclass (detected using

a C++ `dynamic_cast`), then the factory will create a new filter, passing it the `License` for the package's container. If not, it returns `nullptr` to indicate that this filter does not apply to any content within the given `Package`.

Note that the priority assigned to our filter is `1000`. This is the highest priority, and ensures that our filter operates before any other filters get to see the data.

C.6 Requesting Credentials

The `ContentModuleManager` object provides a method, `RequestInputCredentials()` (cf. Section 3.6.5), through which a content module can present a request to the user that they enter their authentication credentials. This process can be simple or complex depending on the implementation of the underlying module-specific credential system. In the case of this example, we use a more complicated setup.

The information presented to the user includes a mandatory title and informational message, a (possibly) pre-populated text input field for the user's account name, and a masked-input text field for the user's password. It then provides a button which allows the user to initiate password reset. The host SDK will ensure that both 'ok' and 'cancel' buttons are included on your behalf– you don't need to add those yourself. All of these are specified using an instance of the `CredentialRequest` class.

The `RequestCredentialInput()` method takes the credential request and returns an `async_result` through which the resulting credentials can be obtained. Even if the user cancels the dialog, a set of credentials corresponding to the module-defined input fields is guaranteed to be returned.

In the event of a cancellation, the implementation will throw a `std::system_error` exception with an error code value of `std::errc::operation_canceled` and a category of `std::generic_category`. This exception is captured by the `std::promise` and is re-thrown by the call to the `async_result` object's `get()` method. The example handles this particular case internally and re-throws or ignores any other exceptions.

The full code listing is presented here; we will break this down for discussion below.

```
std::string SampleDRMModule::RequestCredentials(std::string& account)
{
    // Create a new CredentialRequest object.
    CredentialRequest req("Sign In", "Please enter your credentials to open this book↩
        .");

    // Request has a title & message from the constructor, so add the two text entry ↩
        fields.

    // First is the account name. User can fill this in, but we *might* have a value
    //  from the license-- so we can use that to pre-populate the field.
    req.AddCredential("Account Name", false, account);

    // Add the password field, marking it 'secret' so text entered will not be shown
    //  on screen-- it's up to the OS & UI to determine exactly how that happens.
    req.AddCredential("Password", true);
```

```
    // Maybe the user has forgotten their password-- let's give them a button to use
    //  to reset their password in that case.
    req.AddButton("Forgot Password?", [](const CredentialRequest* request,
                                         std::size_t buttonIndex) {
        // If you're handling multiple buttons with the same function, you can check
        //  which one caused the callback:
        if (request->GetItemTitle(buttonIndex) != "Forgot Password?")
            return;

        // Respond to the button press -- you CAN block here, you're guaranteed to
        //  be called from a background execution context, not the UI context.

        // This here is the dance you can do with std::promise and std::future to
        //  wait for a callback to occur.

        // Create a std::promise, used to set the value.
        std::promise<void> setComplete;

        // Get the std::future from the promise, used to wait for & receive the
        //  value.
        std::future<void> isComplete = setComplete.get_future();

        // Call the async method, passing in a callback which references the
        //  promise.
        // Note that std::promise is movable, not copyable, and C++11 lambdas
        //  apparently don't do implicit moves. C++14 will allow for this using
        //  'generalized lambda captures', though it's only supported in GCC 4.9 so
        //  far. You can use the following macro to test for it:
#if EPUB_COMPILER_SUPPORTS(CXX_INIT_CAPTURE)
        // We have support for naming and initializing lambda capture variables, so
        //  let's use that to move-capture the promise object:
        auto forgotPasswordCallback = [complete(std::move(setComplete))]() {
            // Setting the value will cause any threads waiting on the future to
            //  wake
            complete.set_value();
        };
#else
        // We don't have the ability to move-capture the promise object, so capture
        //  by reference instead. This is ONLY safe because we are blocking in the
        //  same stack context as the promise object; otherwise, we'd have to
        //  declare a std::shared_ptr<std::promise<bool>>, or allocate on the heap
        //  and delete() it from the callback (but what if the callback is never
        //  called?!)
        auto forgotPasswordCallback = [&setComplete]() {
            // Setting the value will cause any threads waiting on the future to
            //  wake
            setComplete.set_value();
        };
#endif
        // Call the implementation function and pass our callback.
```

```
    // Note that the user might have entered a different account name, so we
    //  read that from the credential request.
    std::string account;

    // We can always read the credentials from the request object-- they're not
    //  guaranteed to be complete, however.

    // Because we're getting a const reference to the credentials, we can't use
    //  std::map's array-subscript operator, which is non-const. Instead, we
    //  must use find().
    const Credentials& creds = request->GetCredentials();
    auto item = creds.find("Account Name");
    if (item != creds.end())
        account = item->second.stl_str();

    License::ForgotPassword(account, forgotPasswordCallback);

    // Now wait for the async action to complete. In this instance, we don't
    //  actually care about the result, just that it's complete.
    isComplete.wait();

    // All done-- return from the function now.
});

// Now ask for the credentials to be evaluated. We pass our request into the
//  ContentModuleManager, which implements our interface to the host SDK's
//  interaction facilities:
auto getCreds = ContentModuleManager::Instance()->RequestCredentialInput(req);

// Wait for the credentials to become available, and fetch them.
// Note that if an exception occurred while fetching the credentials, it will be
//  re-thrown at this point; an exception is guaranteed if the user canceled
//  the request.
Credentials creds;
try
{
    creds = getCreds.get();
}
catch (std::system_error& err)
{
    if (err.code().category() == std::generic_category() &&
        err.code().value() == int(std::errc::operation_canceled))
    {
        // the user canceled -- return cleanly
        account.erase();
        return std::string();
    }

    // anything else gets re-thrown
    throw;
```

```
    }
    // all other exceptions bubble up beyond here

    // At this point we're guaranteed that the user clicked 'OK'.

    // Extract two values: the account name and the password.

    // Account returns by-reference:
    account = creds["Account Name"].stl_str();

    // Password returns directly:
    return creds["Password"].stl_str();
}
```

C.6.1 `CredentialRequest` Setup

The first task is to create the `CredentialRequest`. Its constructor takes the title and message, and we use the `AddCredential()` (cf. Section 3.7.7) method to add both the account and password input fields:

```
// Create a new CredentialRequest object.
CredentialRequest req("Sign In", "Please enter your credentials to open this book.");

// Request has a title & message from the constructor, so add the two text entry ↩
     fields.

// First is the account name. User can fill this in, but we *might* have a value
//  from the license-- so we can use that to pre-populate the field.
req.AddCredential("Account Name", false, account);

// Add the password field, marking it 'secret' so text entered will not be shown
//  on screen-- it's up to the OS & UI to determine exactly how that happens.
req.AddCredential("Password", true);
```

C.6.2 Implementing a Button

As noted above, we intend to provide the user with the ability to reset their password. We will implement this by adding a custom button to the credential request, titled 'Forgot Password?', and we will process it using a lambda function defined inline.

The basic form of the call to `AddButton()` (cf. Section 3.7.8) is fairly simple; it takes the button's title[2] and a function object which will be called when the button is pressed. The function MUST conform to the definition laid out in Section 3.7.3.

The button handler is guaranteed to be called in a background execution context, so it is safe to block within it.

[2]Unlike the example, we recommend that you provide an appropriately localized value.

```
// Maybe the user has forgotten their password-- let's give them a button to use
//  to reset their password in that case.
req.AddButton("Forgot Password?", [](const CredentialRequest* request,
                                     std::size_t buttonIndex) {
    «lambda content elided»
});
```

The lambda receives two parameters: a pointer to the `CredentialRequest` to which the button belongs, and the index of the button that was pressed. This allows you to use a single callback to handle multiple buttons and even multiple requests.

If you're handling multiple buttons, you can either keep track of the index value returned from `AddButton()`, or you can ask for the title of the pressed button to determine what action to take:

```
if (request->GetItemTitle(buttonIndex) != "Forgot Password?")
    return;
```

In our example, we use an asynchronous callback-based function provided by the DRM system to run the password reset operation. Since we are guaranteed to run in a background execution context, we're again going to wrap the operation using a promise/future pair, as we did in Section C.4.4:

```
// Create a std::promise, used to set the value.
std::promise<void> setComplete;

// Get the std::future from the promise, used to wait for & receive the
//  value.
std::future<void> isComplete = setComplete.get_future();
```

To correctly handle the `std::promise` object, we want to pass it to the lambda function we're using as the callback for the DRM system's `ForgotPassword()` method. However, since promise objects are only *movable* and not *copyable*, we have a problem with how to do so.

The safest method is to *move* the promise into the lambda, but this requires the use of a C++14 feature called 'generalized lambda captures'. This allows the naming and assignment of variables within a lambda's capture block:

```
auto forgotPasswordCallback = [complete(std::move(setComplete))]() {
    // Setting the value will cause any threads waiting on the future to
    //  wake
    complete.set_value();
};
```

If that functionality is not available, the only option is to capture the promise by reference. This requires that the promise remain in scope until the lambda is executed. In our case this is guaranteed to happen (we are waiting on the future in the same scope), so we can use the alternative format:

```
auto forgotPasswordCallback = [&setComplete]() {
```

```
    // Setting the value will cause any threads waiting on the future to
    //  wake
    setComplete.set_value();
};
```

Note that the host SDK provides a preprocessor macro to test for the relevant compiler support: `EPUB_COMPILER_SUPPORTS(CXX`

Next we obtain the account name from the passed-in `CredentialRequest` and pass that, along with our lambda, to the DRM system's implementation:

```
// Because we're getting a const reference to the credentials, we can't use
//  std::map's array-subscript operator, which is non-const. Instead, we
//  must use find().
const Credentials& creds = request->GetCredentials();
auto item = creds.find("Account Name");
if (item != creds.end())
    account = item->second.stl_str();

License::ForgotPassword(account, forgotPasswordCallback);
```

The only remaining step is to wait for the operation to complete, for which we use our `std::future` object. After this, we return control to the host SDK so the user can enter their updated credentials to complete the request:

```
// Now wait for the async action to complete. In this instance, we don't
//  actually care about the result, just that it's complete.
isComplete.wait();

// All done-- return from the function now.
```

C.6.3 Handling Credential Input

We've now installed two text fields and a button, complete with a fairly complex handler. Now we're ready to fire off the credential request itself and get ready to receive the results:

```
// Now ask for the credentials to be evaluated. We pass our request into the
//  ContentModuleManager, which implements our interface to the host SDK's
//  interaction facilities:
auto getCreds = ContentModuleManager::Instance()->RequestCredentialInput(req);
```

That call returns an `async_result` object that we'll use to fetch the user's credentials once they become available. If the user tries to reset their password, that will all happen for us in the background thanks to our button handler.

When the user presses the 'OK' or 'Cancel' buttons, the `async_result` gets notified in one of two ways:

- If the user **Cancels** the operation, an exception will be thrown for us to catch. The exception is a `std::system_error` instance containing an error code of `std::errc::operation_canceled`.
- If the user hits **OK**, the credentials will be returned to us as a `Credentials` object. Each credential's value is accessed using the name of an input field as a key.

Since one of the results throws an exception, we need to wrap the call in a `try` block:

```
Credentials creds;
try
{
    creds = getCreds.get();
}
```

This call will block until the user presses either the 'OK' or 'Cancel' buttons. In the former case, the credentials will be stored in the `creds` variable. In the latter case, an exception will be caught by the following `catch` clause:

```
catch (std::system_error& err)
{
    if (err.code().category() == std::generic_category() &&
        err.code().value() == int(std::errc::operation_canceled))
    {
        // the user canceled -- return cleanly
        account.erase();
        return std::string();
    }

    // anything else gets re-thrown
    throw;
}
```

In this block, we catch only instances of `std::system_error`, and we check whether the caught exception has the category and error code indicating that the user canceled the operation. This example handles this case by returning empty values for both the account and password components.

If the exception doesn't match this, something else has gone wrong, so the exception is re-thrown for handling elsewhere. No other catch statements are included, so any other types of exception continue to bubble up the stack.

The last step is to extract the account name and password from the `Credentials` object. The account name is retrieved because the user may have entered a different name than the one we obtained from the token file; it is assumed that the DRM system supports this.

The account is returned by reference, and the password is returned directly:

```
    // Account returns by-reference:
    account = creds["Account Name"].stl_str();
```

```
    // Password returns directly:
    return creds["Password"].stl_str();
```

With that done, we can now look at the implementation of our `ContentFilter` subclass.

C.7 Content Filter Implementation

Our content filter accepts a `License` object when it is created, which is then used to initialize a `Decryptor` DRM system object. This is all done inside the object's constructor:

```
SampleDRMDecryptor(const License& license)
    : ePub3::ContentFilter(&NeedsDecryption), mDecryptor(license)
    {}
```

Our content filter operates at priority `1000`, ensuring it is the first such filter to access the raw file data. However, we only encrypt certain types of content; in particular, we never encrypt font files (they might use EPUB3 font obfuscation, however, in which case the host SDK's built-in filters handle them).

Our type-sniffer function, then, needs to inspect each manifest item it's given and determine whether it should apply. It does this by looking for any item with a font media-type, using the same regular expression used by the host SDK's font obfuscation filter. If the manifest item's media type matches, then the type-sniffer function will return false, and the filter will not be applied to this item's data. Otherwise, it returns true, and it will be applied to the data as it is retrieved.

```
bool SampleDRMDecryptor::NeedsDecryption(ePub3::ConstManifestItemPtr item)
{
    // We'll use a regex to check the content type.
    // GNU libstdc++ doesn't implement C++11's std::regex, so Readium SDK has a handy↩
          macro:
    static const REGEX_NS::regex
      TypeCheck("(?:font/.*|application/(?:x-font-.*|vnd.ms-(?:opentype|fontobject)))↩
           ");

    if (REGEX_NS::regex_match(item->MediaType().stl_str(), TypeCheck))
        return false;       // it's a font, so we don't do anything

    // everything else is to be decrypted
    return true;
}
```

The decryption method itself is simple: the input data is simply passed directly to the underlying cryptography engine. Note that some cryptography API's don't work with piecemeal data and expect to receive everything in one go.[3] In that case, you should override the `RequiresCompleteData()` method in your content filter to ensure that you are given all the data in a single pass.

The implementation of this module's `FilterData()` method is shown below.

[3] In particular, Microsoft's WinRT Windows::Security APIs.

```
void* SampleDRMDecryptor::FilterData(FilterContext * /* ignored / unused */,
                                     void *data, size_t len, size_t *outputLen)
{
    // we haven't overridden RequiresCompleteData() to return true, so we're going
    //  to receive piecemeal data. Our decryption routines are fine with that.

    // a nice & simple pass-through
    return mDecryptor.DecryptBlock(data, len, outputLen);
}
```

Normative References

[RFC2119] *Key Words for use in RFCs to indicate Requirement Levels.*
S. Bradner, Internet RFC 2119, March 1997.
http://www.ietf.org/rfc/rfc2119.txt

[EPUB3] *EPUB 3.0 Overview.*
G. Conboy et al. 11 October 2011.
http://idpf.org/epub/30/spec/

[EPUB-CFI] *EPUB Canonical Fragment Identifier (epubcfi) Specification.*
P. Sorotokin et al. 11 October 2011.
http://idpf.org/epub/linking/cfi/epub-cfi.html

[OCF] *EPUB Open Container Format (OCF) 3.0.*
J. Pritchett, M. Gylling. 11 October 2011.
http://idpf.org/epub/30/spec/epub30-ocf-20111011.html

[OPF] *EPUB Publications 3.0.*
M. Gylling, W. McCoy, M. Garrish. 11 October 2011.
http://idpf.org/epub/30/spec/epub30-publications-20111011.html

[EPUB3-MO] *EPUB Media Overlays 3.0.*
M. DeMeglio, D. Weck. 11 October 2011.
http://www.idpf.org/epub/30/spec/epub30-mediaoverlays.html

[EPUB3-FO] *EPUB Font Obfuscation.*
J. Pritchett, M. Gylling. 11 October 2011.
http://idpf.org/epub/30/spec/epub30-ocf-20111011.html#
font-obfuscation

[EPUB-SW] *EPUB3 Content Switching.*
M. Gylling, W. McCoy, E. Etemad, M. Garrish. 11 October 2011.
http://www.idpf.org/epub/30/spec/epub30-contentdocs-20111011.html#
sec-xhtml-content-switch

[URI] *Uniform Resource Identifiers (URI): generic syntax.*
T. Berners-Lee, R. Fielding, L. Masinter. Internet RFC 3986. January 2005.
http://www.ietf.org/rfc/rfc3986.txt

[C++11] *Programming Languages – C++.*
P. Becker et al. ISO/IEC Standard 14882:2011. 11 April 2011.
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.
htm?csnumber=50372

[XML-DSig] *XML-Signature Syntax and Processing Version 1.1.*
M. Bartel et al. 3 March 2011.
http://www.w3.org/TR/xmldsig-core1/

[XML-ENC] *XML Encryption Syntax and Processing Version 1.1.*
D. Eastlake et al. 3 March 2011.
http://www.w3.org/TR/xmlenc-core1/

Informative References

[SDK] *Readium SDK Project Overview.*
The Readium Foundation Members. 2013.
http://readium.org/projects/readium-sdk/

[LCP] *EPUB Lightweight Content Protection.*
J. Dovey. 5 September 2012.
https://dl.dropboxusercontent.com/u/896638/Kobo-EPUB-LCP.pdf

[std::future] *CppReference.com.*
cppreference.com contributors. 2013.
http://en.cppreference.com/w/cpp/thread/future

[FutProm] *en.wikipedia.org.*
Wikipedia contributors. 2013.
http://en.wikipedia.org/wiki/Futures_and_promises

[CXX-REF] *CppReference.com.*
cppreference.com contributors. 2013.
http://en.cppreference.com/w/cpp/language/reference

[CXX17-FUT] *Improvements to* `std::future<T>` *and Related APIs.*
N. Gustafsson, A. Laksberg, H. Sutter, S. Mithani. 27 September 2013.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3784.pdf

[ShPtr] *CppReference.com.*
cppreference.com contributors. 2013.
http://en.cppreference.com/w/cpp/thread/future

[ShFromThis] *CppReference.com.*
cppreference.com contributors. 2013.
http://en.cppreference.com/w/cpp/memory/enable_shared_from_this/shared_from_this

Index

DRMInterface.h

```
//
//  DRMInterface.h
//  ePub3
//
//  Created by Jim Dovey on 11/8/2013.
//  Copyright (c) 2013 The Readium Foundation and contributors. All rights reserved.
//

#ifndef ePub3_DRMInterface_h
#define ePub3_DRMInterface_h

#include <string>
#include <functional>

class License
{
public:
    License(void* bytes, std::size_t len);
    License(const License&);
    ~License();

    License& operator=(const License&);

    const std::string& GetContentIdentifier() const;
    const std::string& GetContentURL() const;

};

class Decryptor
{
public:
    Decryptor(const License& license);
    Decryptor(const Decryptor&);
    ~Decryptor();

    Decryptor& operator=(const Decryptor&);

    // may return data in a buffer allocated with operator new[], or in input buffer
    void* DecryptBlock(void* bytes, std::size_t len, std::size_t *decryptedLen);

};

class Token
{
public:
    Token(const std::string& path);
    ~Token();

    bool NeedsCredentials() const;

    const std::string& GetAccountName() const;
    bool VerifyPassword(const std::string& password);
    bool VerifySignature(void* sigBytes, std::size_t sigLen);

    License RetrieveContentLicense() const;

};

class Downloader
{
public:
    typedef std::function<void(const std::string& path)>    CompletionFn;

public:
    Downloader(const std::string& contentURL);
    ~Downloader();
```

```
    bool InitiateDownload(CompletionFn completion);

};

class ContentStore
{
protected:
    ContentStore();

public:
    ~ContentStore();

    static ContentStore* Instance();

    bool HasLocalContent(const License& license);

    std::string PathToLocalContent(const License& license);
    std::string StoreLocalContent(const License& license, const std::string& localPath);

};

#endif
```

DRMModule.h

```
//
//  DRMModuleSample.h
//  ePub3
//
//  Created by Jim Dovey on 11/8/2013.
//  Copyright (c) 2013 The Readium Foundation and contributors. All rights reserved.
//

#ifndef __ePub3__DRMModuleSample__
#define __ePub3__DRMModuleSample__

#include "DRMInterface.h"
#include <ePub3/epub3.h>
#include <ePub3/content_module.h>
#include <ePub3/filter.h>

class SampleDRMModule : public ePub3::ContentModule
{
public:
    SampleDRMModule();
    virtual ~SampleDRMModule() {}

    ////////////////////////////////////////////
    // Token files

    virtual
    ePub3::async_result<ePub3::ContainerPtr>
    ProcessFile(const ePub3::string& path,
                std::launch policy=std::launch::any);

    ////////////////////////////////////////////
    // Content Filters

    virtual
    void
    RegisterContentFilters();

    ////////////////////////////////////////////
    // User actions

    virtual
    ePub3::async_result<bool>
```

```
    ApproveUserAction(const ePub3::UserAction& action);

    ////////////////////////////////////////////
    // Internal

    static void Initialize();

    const License& LicenseForContainer(ePub3::ContainerPtr& container);

private:
    bool    mFiltersRegistered;
    std::map<ePub3::ContainerPtr, License, std::owner_less<ePub3::ContainerPtr>>
mLicenses;

};

class SampleDRMDecryptor : public ePub3::ContentFilter
{
protected:
    static bool NeedsDecryption(ePub3::ConstManifestItemPtr item);

public:
    SampleDRMDecryptor(const License& license) : ePub3::ContentFilter(&NeedsDecryption),
mDecryptor(license) {}
    virtual ~SampleDRMDecryptor() {}

    // NB: we encrypt on a container level, so we don't need per-content-document context
info

    virtual void * FilterData(ePub3::FilterContext* context, void *data, size_t len,
size_t *outputLen);

private:
    Decryptor   mDecryptor;

};

#endif /* defined(__ePub3__DRMModuleSample__) */
```

DRModule.cpp

```
//
//  DRMModuleSample.cpp
//  ePub3
//
//  Created by Jim Dovey on 11/8/2013.
//  Copyright (c) 2013 The Readium Foundation and contributors. All rights reserved.
//

#include "DRMModuleSample.h"
#include <ePub3/content_module_manager.h>
#include <ePub3/filter_manager.h>
#include <ePub3/container.h>
#include <ePub3/package.h>
#include <future>
#include <memory>

// not all platforms have std::regex, so Readium SDK will define this to be <regex>
//  or <boost/regex.hpp> (provided) as necessary
#include REGEX_INCLUDE

using namespace ePub3;

SampleDRMModule::SampleDRMModule()
    : mFiltersRegistered(false), mLicenses()
{
```

```
}

void SampleDRMModule::Initialize()
{
    auto myModule = std::make_shared<SampleDRMModule>();
    ContentModuleManager::Instance()->RegisterContentModule(myModule, "SampleDRM");
}

async_result<ContainerPtr>
SampleDRMModule::ProcessFile(const string& path, std::launch policy)
{
    Token* token = new Token(path.stl_str());
    if (token == nullptr)
    {
        // the constructor didn't recognize the file & threw an exception

        // this is the roundabout way C++11 gives us of returning an initialized
std::future
        promised_result<ContainerPtr> promise;
        promise.set_value(nullptr);
        return promise.get_future();
    }

    // start an async task, for example using C++11's std::async()
    return std::async(policy, [this, token]() -> ContainerPtr {
        if (token->NeedsCredentials())
        {
            std::string account = token->GetAccountName();
            std::string password;

            // <<request password>>

            if (token->VerifyPassword(password) == false)
                return nullptr;     // failed to authorise access to the content
        }

        License license = token->RetrieveContentLicense();

        std::string localPath;
        ContentStore* store = ContentStore::Instance();
        if (store->HasLocalContent(license))
        {
            localPath = store->PathToLocalContent(license);
        }
        else
        {
            Downloader downloader(license.GetContentURL());
            std::promise<std::string> downloadPromise;
            std::future<std::string> downloadFuture = downloadPromise.get_future();

            // fire off the download using its built-in async API
            downloader.InitiateDownload([&downloadPromise](const std::string&
downloadedPath) {
                downloadPromise.set_value(downloadedPath);
            });

            // already running in the background here, so block until the download is
done
            std::string downloadPath = downloadFuture.get();

            // place the downloaded item into our local store
            localPath = store->StoreLocalContent(license, downloadPath);
        }

        // we have the path to the actual EPUB file now, so open it
        // we use the internal OpenContainerSync() method, which doesn't attempt to use
ContentModules
        ContainerPtr result = Container::OpenContainerSync(localPath);
```

```
        // tell the container that we created it-- this helps our filter factory
determine if it's needed
        result->SetCreator(shared_from_this());

        // store the container/license pair here so we can retrieve the license for our
filter(s) later
#if EPUB_HAVE(CXX_MAP_EMPLACE)
        mLicenses.emplace(result, license);
#else
        // GNU libstdc++ doesn't implement emplace() on std::map as of GCC 4.8
        mLicenses[result] = license;        // License has a default constructor, so we
can do this
//        mLicenses.insert(std::make_pair(result, license));      // if License wasn't
default-constructible we'd use this
#endif

        // lastly, return the pointer -- std::async() takes care of the promise/future
for us
        return result;
    });
}

void
SampleDRMModule::RegisterContentFilters()
{
    if (mFiltersRegistered)
        return;

    FilterManager::Instance()->RegisterFilter("SampleDRMDecryptor", 1000,
[this](ConstPackagePtr pkg) -> ContentFilterPtr {
        // find the container.
        ContainerPtr container = pkg->Owner();

        // fetch the ContentModule which created the container, if any.
        std::shared_ptr<ContentModule> module = container->Creator();

        // if the creator is an instance of the correct class...
        auto myModule = std::dynamic_pointer_cast<SampleDRMModule>(module);
        if (bool(myModule))
        {
            // ...query it for the license for this container and return a filter.
            return std::make_shared<SampleDRMDecryptor>(myModule-
>LicenseForContainer(container));
        }
        else
        {
            // ... otherwise, this package isn't ours, so we don't provide a filter for
it.
            return nullptr;
        }
    });
}

////////////////////////////////////////////////////////////////////////////////

bool SampleDRMDecryptor::NeedsDecryption(ePub3::ConstManifestItemPtr item)
{
    // remember that the filter is only allocated for packages loaded by our module
    //  so we don't need to check that here.

    // We don't encrypt fonts (though they may use EPUB3 font obfuscation), so ignore
those

    // We'll use a regex to check the content type.
    // GNU libstdc++ doesn't implement C++11's std::regex, so Readium SDK has a handy
macro:
    static const REGEX_NS::regex TypeCheck("(?:font/.*|application/(?:x-font-.*|vnd.ms-
```

```
(?:opentype|fontobject)))");

    if (REGEX_NS::regex_match(item->MediaType().stl_str(), TypeCheck))
        return false;       // it's a font, so we don't do anything

    // everything else is to be decrypted
    return true;
}

void* SampleDRMDecryptor::FilterData(FilterContext * /* ignored / unused */, void *data,
size_t len, size_t *outputLen)
{
    // we haven't overridden RequiresCompleteData() to return true, so we're going
    //  to receive piecemeal data. Our decryption routines are fine with that.

    // a nice & simple pass-through
    return mDecryptor.DecryptBlock(data, len, outputLen);
}
```