Core / Modules Componentization

Objective

To make core smaller and simpler for code health and to make it faster to build chrome.
In this document, we will define core and modules from the viewpoint of blink implementation. So “core” means “Source/core/” and “Source/bindings/core”, and “modules” means “Source/modules” and “Source/bindings/modules/”.

Smaller and simpler:

Core should be simple and small for code health. If core has too many code and too complex dependencies, it is very difficult to understand and update (including maintaining existing features, fixing bugs, adding new features and so on) core. It is better to move component which are self-dependent to another place, i.e. modules.

Build faster:

If we need to link too many objects at one time, linker will spend a very long time. To solve this issue, we are now trying to make linker link blink_modules.dll separately from blink_core.dll” (crbug.com/358074). This also means, it is not good to have many code in core. Currently the ratio of core : modules is 6 : 1 (c.f. core object size vs modules object size). If possible, it would be better to move more files from core/ to modules/.

Allowed Dependency

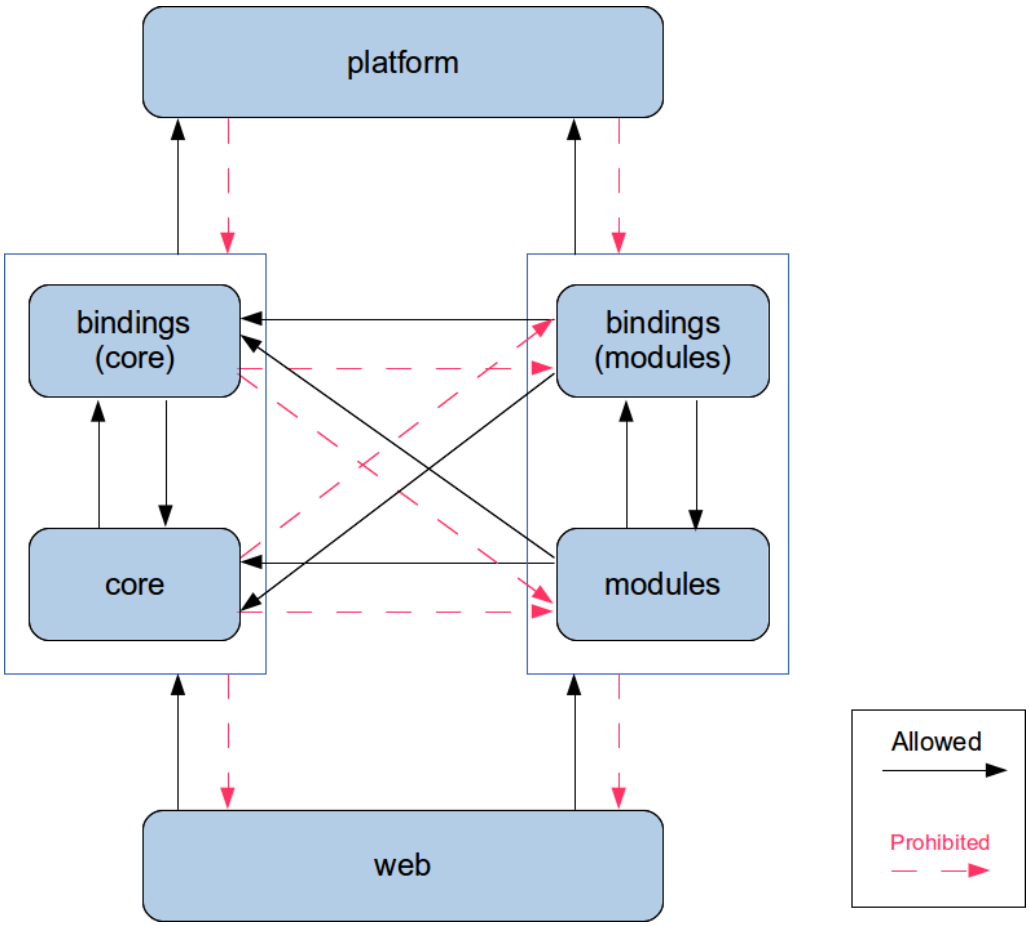
The following table shows current rules about component’s dependency.

Component A	Component B	A can depend on B
core	modules	NO
	web	NO
	platform	YES
modules	core	YES
	web	NO
	platform	YES
web	core	YES
	modules	YES
	platform	YES
platform	core	NO

	modules	NO
	web	NO

Dependency rules

The following figure shows the same rules:



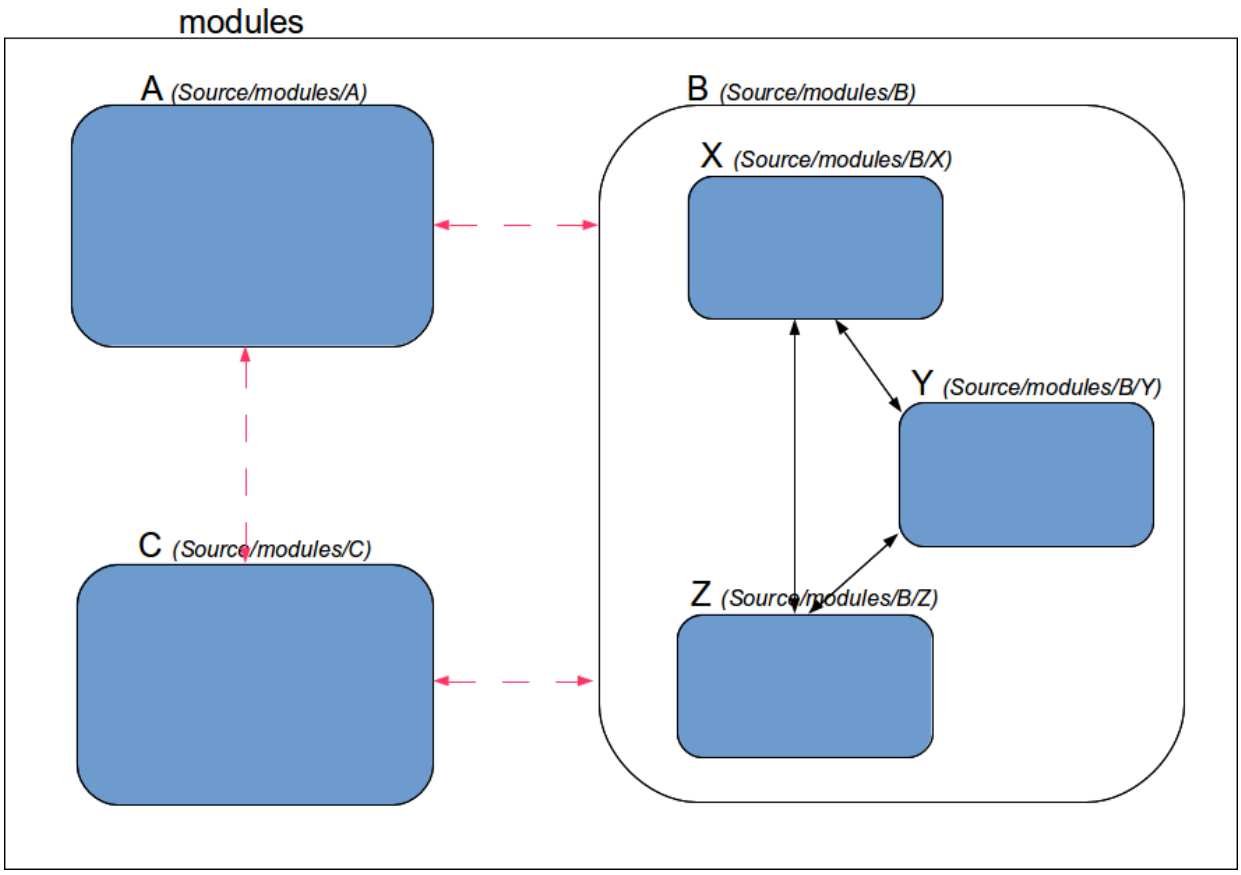
For example, web (Source/web) can use core and modules code (Source/core and Source/modules). However, core and modules cannot use web code. web should only define public Blink interfaces.

Looking at modules, we also have the following rules:

- Each top level module cannot depend on any other top-level modules.
- If some module (e.g. Source/modules/B) depends on Source/modules/X, the X should be placed in Source/modules/B/X.

The following figure shows this rule. We have three top-level modules, i.e. A, B, and C.
A, B, and C cannot depend on one another.

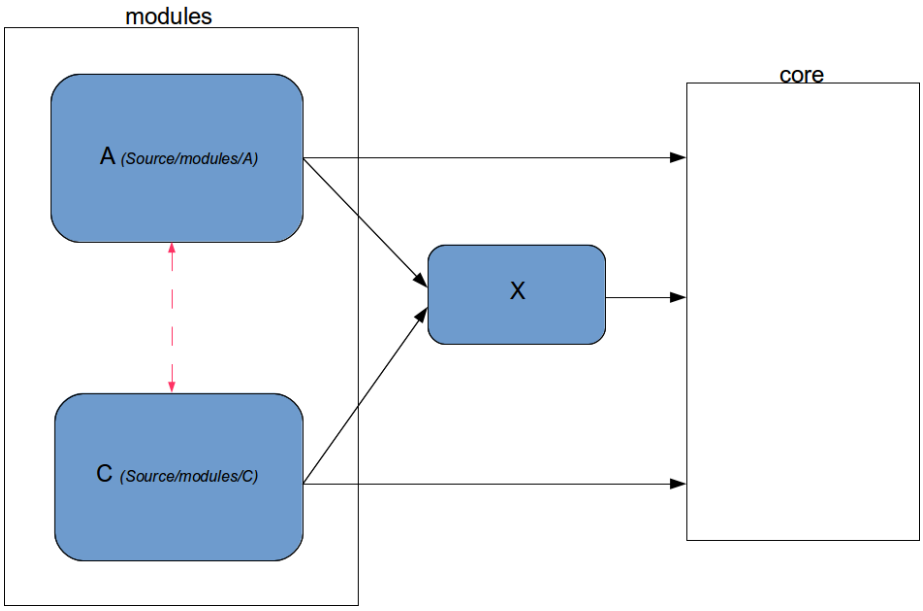
If we want to have some modules which are depend on one another, move the modules under some top-level module, i.e. B/X, B/Y and B/Z.



However, currently some modules break the rule, i.e. webaudio, speech, geofencing, and push_messaging. (c.f. Appendix.)

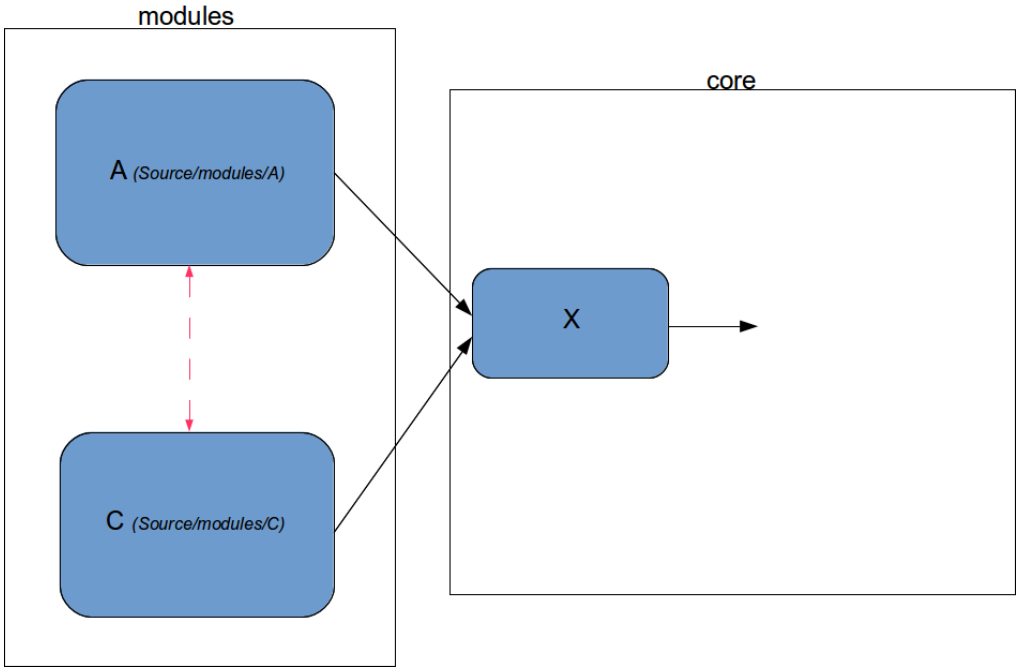
Problem

If many top-level modules depend on some module, i.e. X, where should we place X?

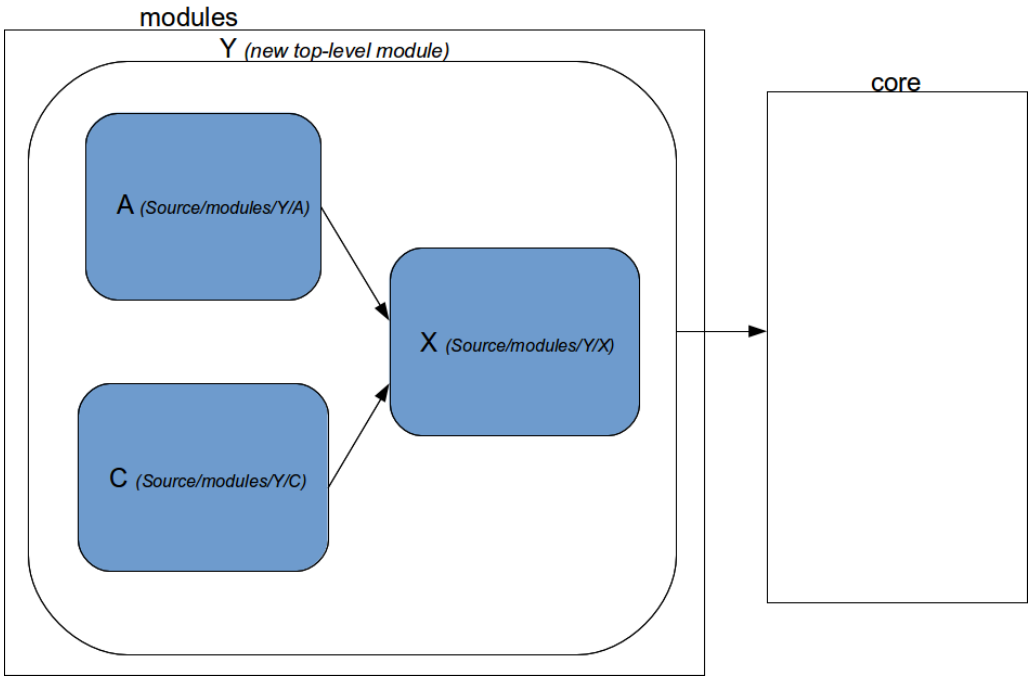


Since we cannot allow dependency among top-level modules, currently we need to do either of the following ways:

1. X should be in core, or



2. X should not be top-level module. Make another top-level module (e.g. Y) and move all dependency modules (e.g. X, A and B) under the top-level module.



If we do (1), when we want to support new specs or to update features according to specs' updates, some of modules might be moved into core. Because specs have neither core nor no modules. So finally most of modules might be moved into core. We will see too large core and very small modules. Probably we don't need modules.

If we do (2), ditto. Finally most of modules will be submodules of one very large module (e.g. Source/modules/all). So we will see core and one "all" module. In this case, we don't need modules. We need only core and the "all".

Solution

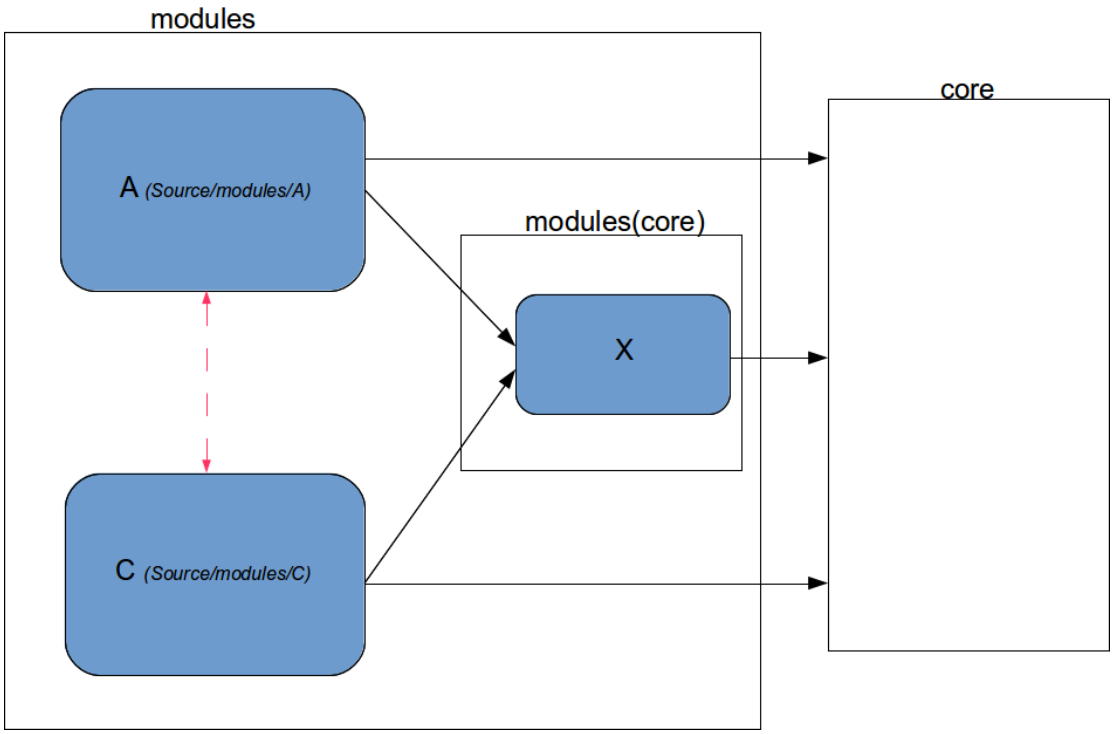
Basic idea

We will change the rule: "top-level modules cannot depend on one another." So we will make it possible for modules to depend on some modules.

Solution 1: add modules(core)

If modules which depend on one another are moved to modules(core), other modules can depend on modules(core).

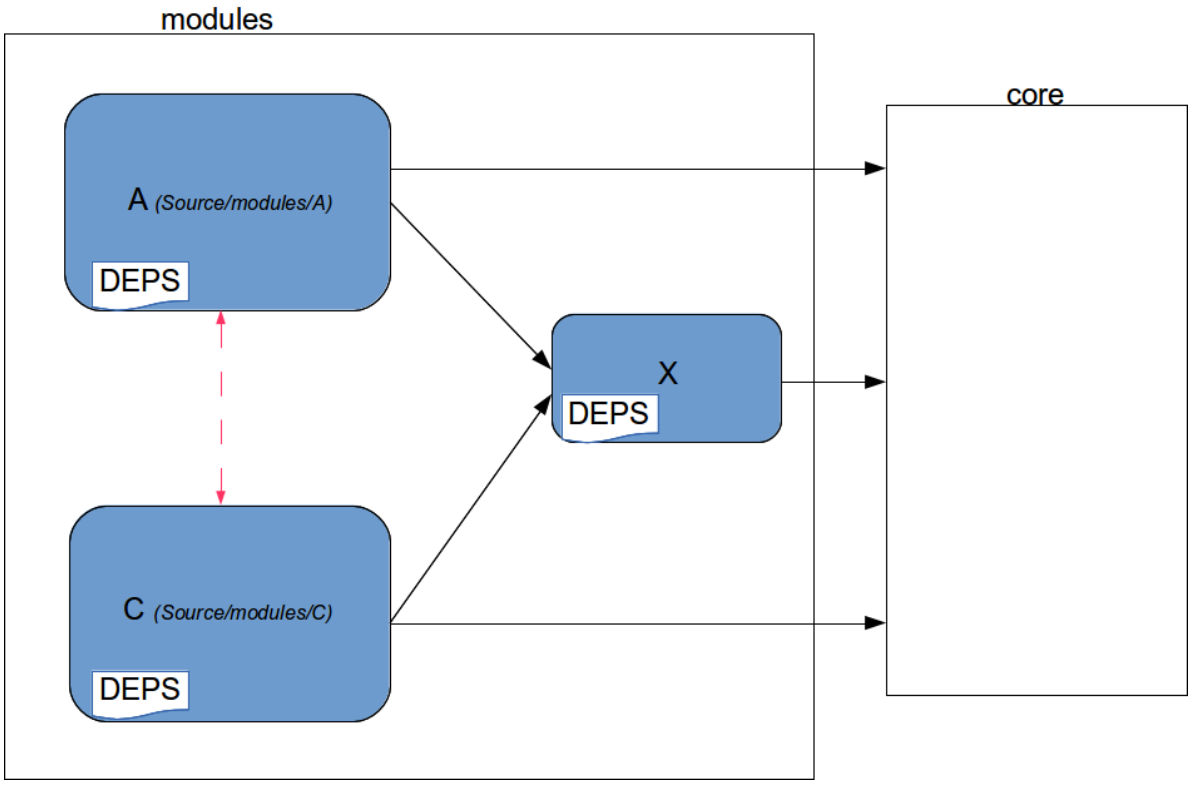
However, modules(core) can depend on only modules(core) and modules(core) cannot depend on the other modules.



Adding modules(core)

[Proposed] Solution 2: just allow modules to depend on one another

Add DEPS to disallow complex dependencies among modules. After adding DEPS to each modules, we should be careful to add new dependencies to the DEPS. (c.f. concept patch for current modules)



Adding DEPS files

Solution 2 is better than Solution 1, because:

- This mirrors Chromium's //components implementation.
- It is possible to avoid complex dependencies in modules(core).
- It is easy to understand current dependencies by looking DEPS files.
- We don't need to move files when some module breaks the rule.

Moving Targets from core/ to modules/

The above solution enables us to move more files from core to modules. The followings are primary targets which we try to move:

- core/storage,
- core/streams,
- core/timing

The followings are secondary targets:

- core/accessibility,
- core/clipboard,
- core/plugins,

- core/workers,
- part of core/xml

We will show why the aboves are primary / secondary targets.

core/storage

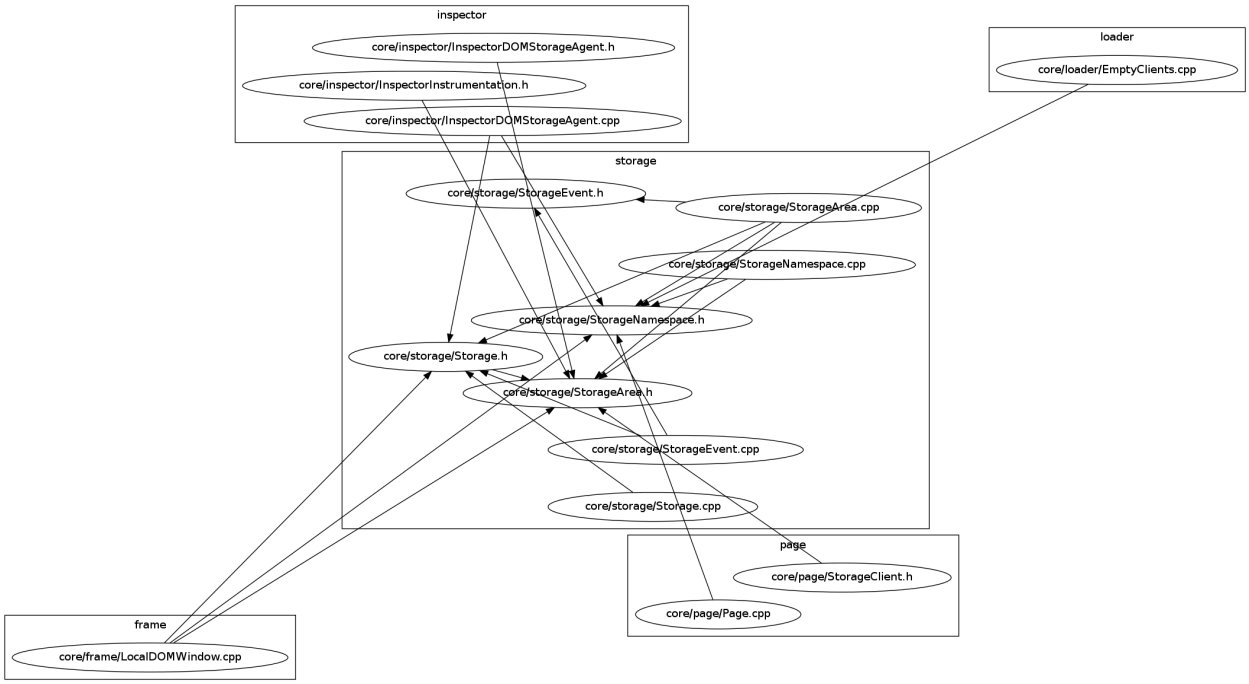


Fig. core/storage #include dependency

The above figure shows dependency among core/storage and others. Small number of core files in inspector, loader, frame and page, depend on core/storage. It would be possible to move storage from core to modules.

core/streams

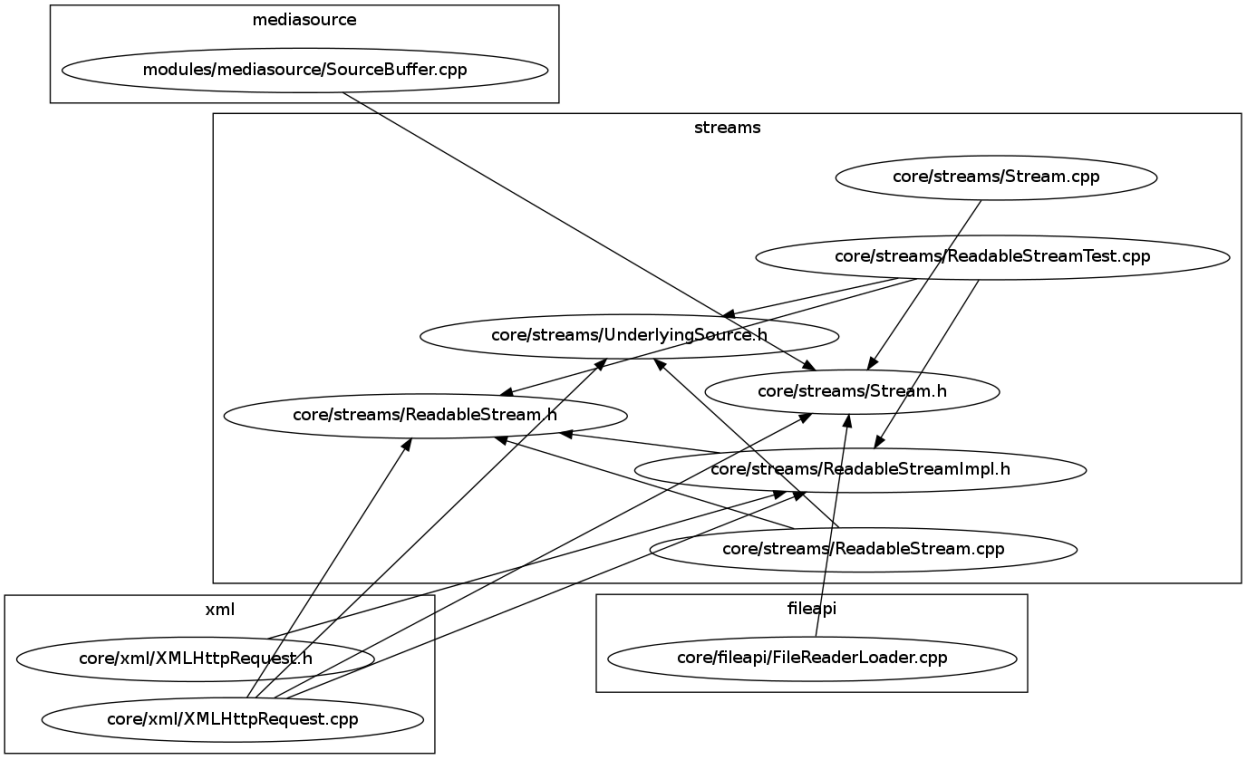


Fig: core/streams dependency

Small number of core files depend on core/streams. If possible, after moving streams from core to modules, update mediasource's DEPS.
Since we are planning to move XMLHttpRequest from core to modules (and we will allow dependencies among modules), we need to fix only FileReaderLoader.cpp.

core/timing

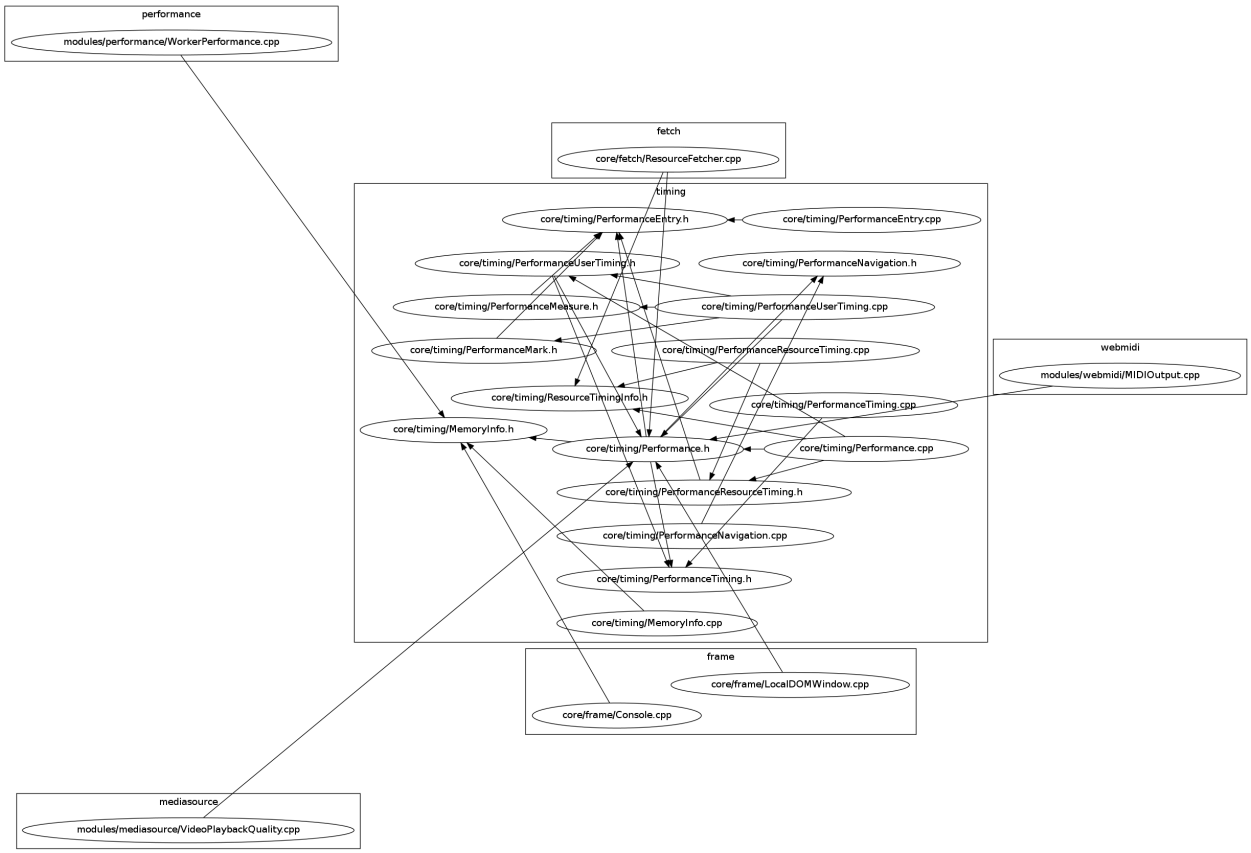
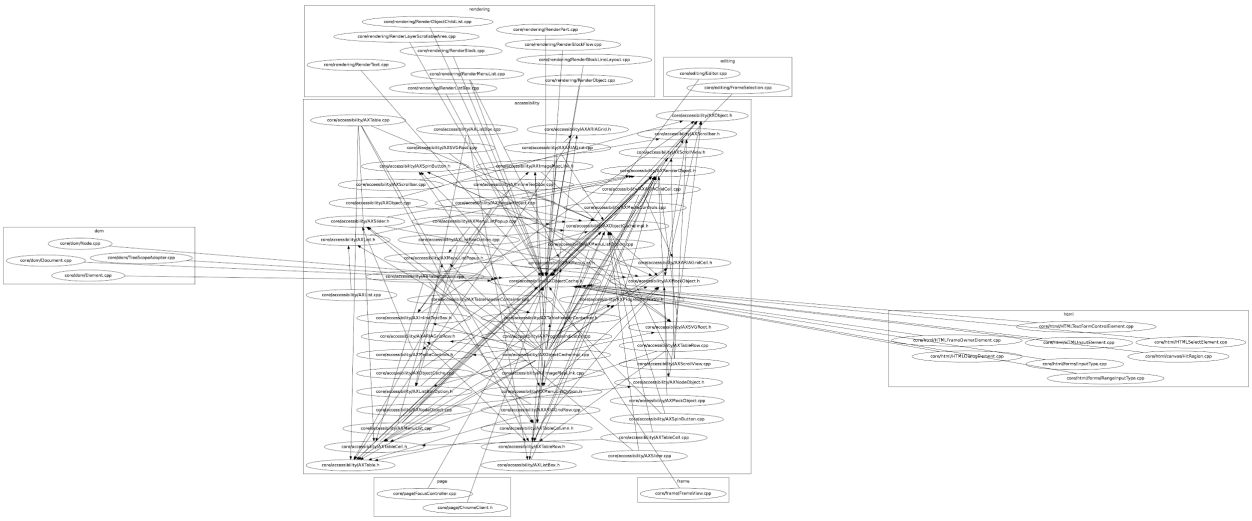


Fig: core/timing dependency

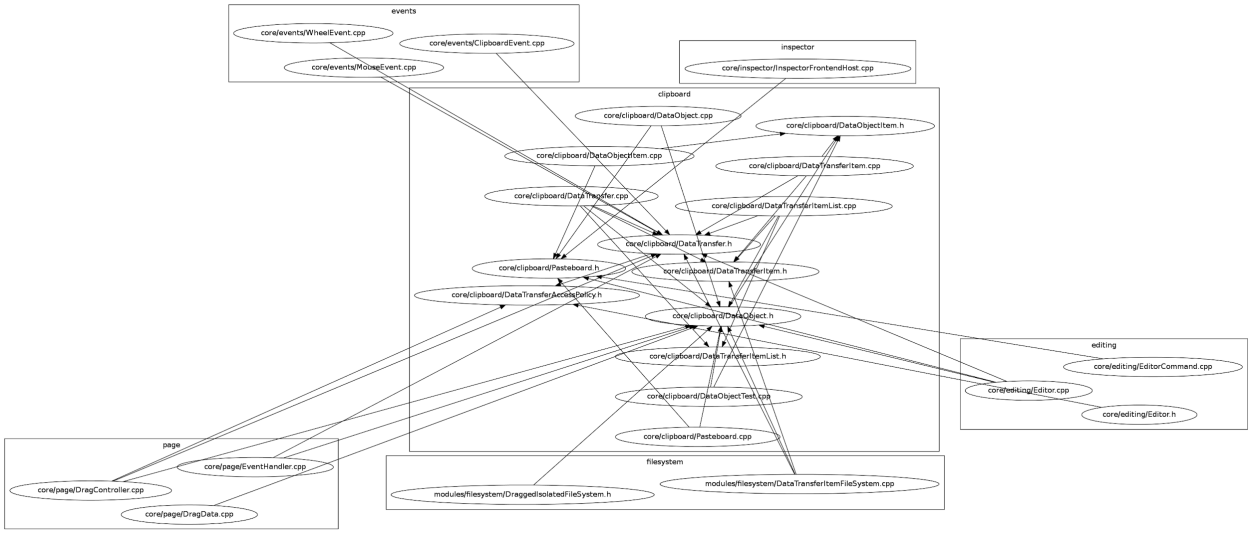
Small number of core files depend on core/timing. It would be possible to move core/timing from core to modules. Since modules/webmidi, modules/mediasource, and modules/performance depend on core/timing, we need update their DEPS files.

core/accessibility



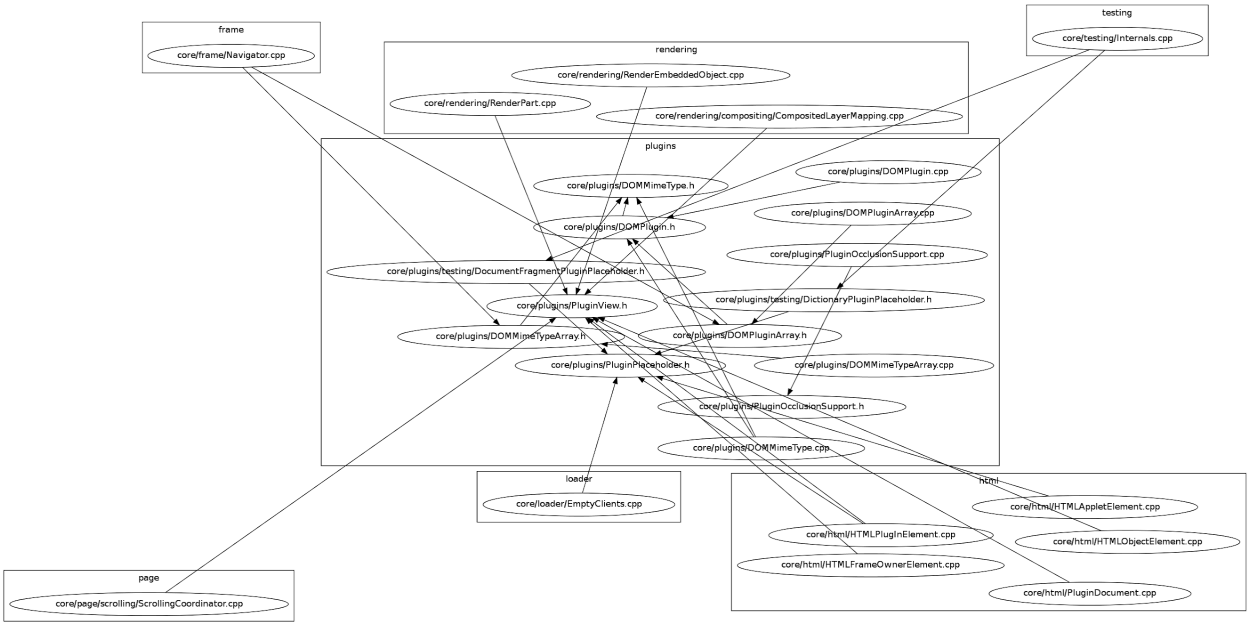
Others depend on AXObjectCache.h. We need to investigate how others use AXObjectCache.h.

core/clipboard



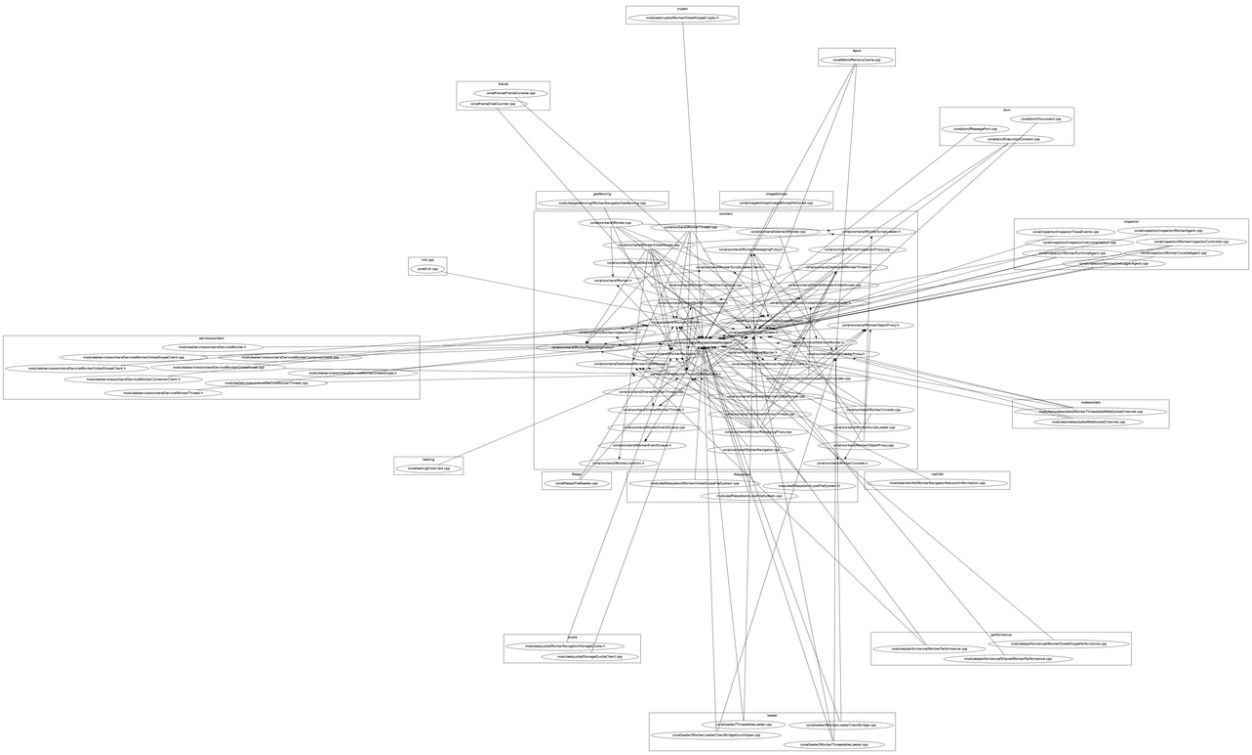
Relatively small number of files depend on core/clipboard.

core/plugins

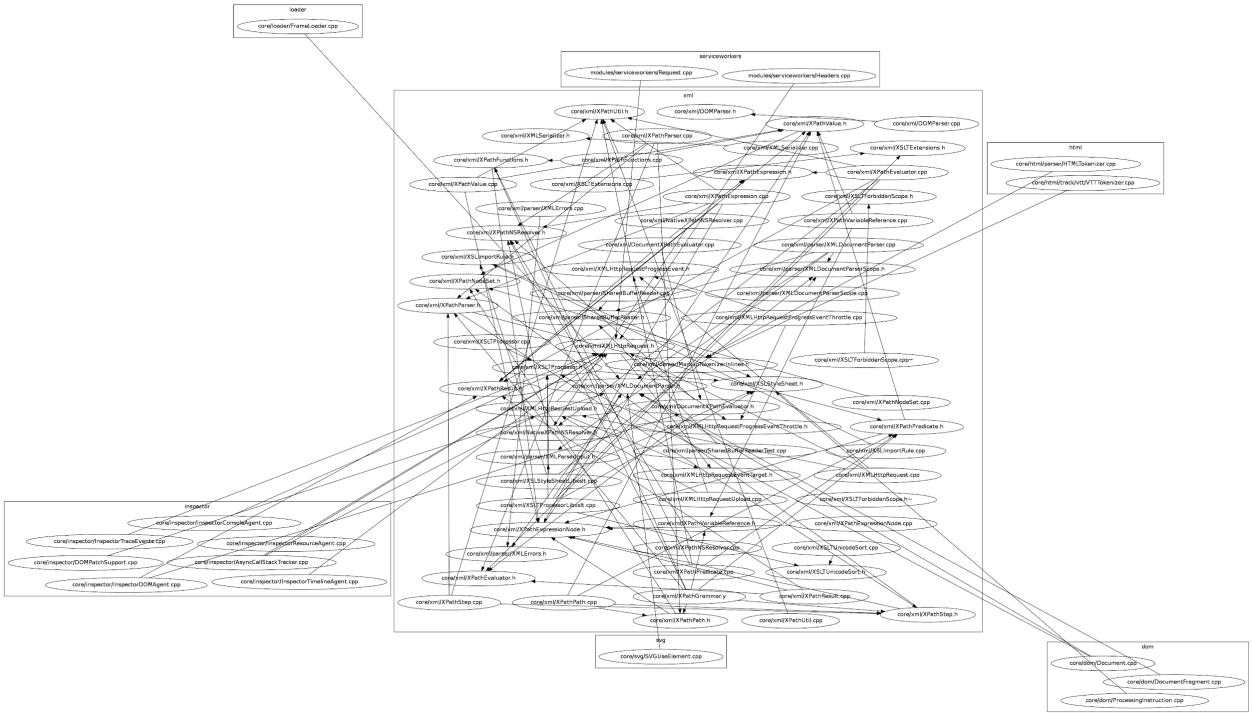


Relatively small number of files depend on core/plugins.

core/workers



core/xml



It would be possible to move XMLHttpRequest, XSLT-related code from core to modules.

It is difficult to move XMLDocumentParser.

How to move files from core to modules

Use partial interface

If a new feature updates some interface defined in core, it would be possible to use partial interface to update the interface and to place the partial interface's IDL file in modules.

For example, WindowQuota.idl uses partial interface to update Window:

```
[
    ImplementedAs=DOMWindowQuota,
] partial interface Window {
    [DeprecateAs=PrefixedStorageInfo] readonly attribute
DeprecatedStorageInfo webkitStorageInfo;
};
```

It is possible to use overloads between interfaces in core and partial interfaces in modules. For example,

```
partial interface URL {
    [CallWith=ExecutionContext] static DOMString?
createObjectURL(MediaStream stream);
};
```

Source/modules/mediastream/URLMediaStream.idl

```
interface URL {
    [RaisesException, CallWith=ExecutionContext] static DOMString?
createObjectURL(Blob? blob);
...
};
```

Source/core/dom/URL.idl

Use union type (and partial interface)

CodeGenerator will support union types between core and modules. So suppose that we have the following:

```
typedef (CORE1 or CORE2 or MODULE1 or MODULE2) TypeX;
```

CORE1 and *CORE2* are interfaces defined in core, and *MODULE1* and *MODULE2* are interfaces defined in modules.

In this case, we will update interfaces, which use TypeX, in core:

```
typedef (CORE1 or CORE2 or MODULE1 or MODULE2) TypeX;

interface TypeXUser {
    attribute TypeX? valueX;
};
```

and CodeGenerator will check dependencies and will generate core code and modules code.

Use Supplementable and Supplement

TBD

Appendix.

Dependencies:

- core/css
- core/dom
- core/events
- core/fetch

- core/frame
- core/html
- core/inspector
- core/loader
- core/page
- core/paint
- core/rendering
- core/svg
- core/imagebitmap - very simple, but Window implements ImageBitmapFactories.