# KeyedServiceFactory design document

This document aims to provide a way for reusing the `KeyedService` factories infrastructure for the Chrome on iOS code.

Currently Chrome on iOS code is in a forked repository of the Chromium repository (aka. downstream repository). There is a project to upstream into the Chromium repository (aka. upstream repository).

The componentization of features was a first step in that direction. However, the upstream code does not include an embedder for iOS. Before such an embedder can be contributed, we need infrastructure to build `KeyedService`.

## iOS constraints

The code targeting iOS has a major restriction in that it cannot use `//content`. This is because `//content` is based on WebKit and v8 but iOS browser must use `UIWebView` or `WKWebView` classes from `UIKit` to render HTML content.

The current `KeyedService` factory infrastructure is based on `content::BrowserContext` and cannot be used on iOS. The only dependencies on `content::BrowserContext` are:

1. `content::BrowserContext::IsOffTheRecord()`,
2. `content::BrowserContext::GetPath()`, only for `DEBUG` build,
3. casting between `content::BrowserContext` and `Profile` (as `Profile` is the only class implementing `content::BrowserContext` interface in `//chrome/browser` code).

The downstream iOS code offer an interface similar to `content::BrowserContext` called `ios::BrowserState`. It is almost ready for upstreaming. It offers the same required interface and has a single implementation in the iOS code, so we can have the same type-safe conversion to the implementation class if needed.

Previous effort showed us that switch day solution, i.e. waiting for everything to be ready before making the transition from one implementation to another, does not work. On the other hand, the ability to gradually transition from one implementation to another by step gets much more traction.

To allow for gradual transitioning, our design needs to allow `BrowserStateKeyedServiceFactories` to depend on `BrowserContextKeyedServiceFactories` and vice versa **during the transition phase only**.

## Proposed Solution: Use Inheritance of Factories

## Overview

Both `ios::BrowserState` and `content::BrowserContext` inherit from `base::SupportsUserData`. Since this class is in base, it is usable from core code of the component; thus, we can have a `KeyedServiceFactory` superclass that deals with `base::SupportsUserData` keys, and have the real classes do the conversion to the correct type with a `static_cast`.

Suggested solution is:

- Implement the inheritance-based design described in detail below upstream, with the contract that upstream `content::BrowserContext` and `ios::BrowserState` factories cannot depend on each other,
- Fork the downstream implementation of `BrowserStateKeyedServiceFactory` to allow cross-dependencies during the transition, this fork is small (only two methods have to be touched),
- Shift the downstream code to start using the `BrowserStateKeyedServiceFactory` for the clients, and upstream them when they only depend on other `ios::BrowserState` factories.

The inheritance based solution has been prototyped in downstream code in 95947013. It is less clean than a template-based solution (see below) but allow us to gradually move from using `BrowserContextKeyedServiceFactory` to using `BrowserStateKeyedServiceFactory`.

## Detailed Design

```
class KeyedServiceFactory;

class DependencyManager {
 protected:
  DependencyManager();
  virtual ~DependencyManager();

  void AddEdge(KeyedServiceFactory* depended,
               KeyedServiceFactory* dependee);

  void CreateContextServices(base::SupportsUserData* context);
  void DestroyContextServices(base::SupportsUserData* context);

  // And so on...
 private:
#ifndef NDEBUG_
  virtual base::FilePath GetPath(base::SupportsUserData* context) const = 0;
#endif
};

class KeyedServiceFactory {
 protected:
  KeyedServiceFactory(const char* name, DependencyManager* manager);
  virtual ~KeyedServiceFactory();

 private:
  virtual base::SupportsUserData* GetContextToUse(
      base::SupportsUserData* context) const = 0;
```

```
  // And so on...
 private:
  DependencyManager* dependency_manager_;
};

class BrowserContextKeyedServiceFactory : public KeyedServiceFactory {
 private:
  virtual content::BrowserContext* GetBrowserContextToUse(
      content::BrowserContext* context) const {
    // Safe default for Incognito mode: no profile.
    if (context->IsOffTheRecord())
      return nullptr;
    return context;
  }

  virtual base::SupportsUserData* GetContextToUse(
      base::SupportsUserData* context) const final {
    // This cast is safe since BrowserContextKeyedService will only ever be used
    // with content::BrowserContext contexts.
    return GetBrowserContextToUse(static_cast<content::BrowserContext*>(context));
  }
};
```

This introduce a slight weakness in the interface, but by making all the method of the base class protected and by sealing the virtual methods used (using the C++11 `final` feature), we can make it as safe as it currently is (i.e. `BrowserContextKeyedServiceFactory` will only deal with `content::BrowserContext*`).

Moreover, in the downstream code, there is a `content::BrowserContext` associated to each `ios::BrowserState` (since downstream code is forked and need a way to use the current factories). With this design, we can have `BrowserContextKeyedServiceFactory` and `BrowserStateKeyedServiceFactory` depends on each other, facilitating the migration of the code in the downstream repository by introducing factories for already componentized features without waiting for the componentization to be complete.

This cross-dependency will only be used in downstream repository while the features are converted. The upstream code will be clean and only use `ios::BrowserState`.

# Other considered designs

## Template-based design

Since `content::BrowserContext` and `ios::BrowserState` offer the same methods required by `KeyedService` factories infrastructure it is possible to templatize the factories on the type of context to use.

```
template <typename Context>
class KeyedServiceFactory;

template <typename Context>
class DependencyManager {
 public:
```

```
  void AddEdge(KeyedServiceFactory<Context>* depended,
               KeyedServiceFactory<Context>* dependee);

  void CreateContextServices(Context* context);
  void DestroyContextServices(Context* context);

  // And so on...
 protected:
  DependencyManager();
  ~DependencyManager();
};

template <typename Context>
class KeyedServiceFactory {
 public:
  void DependsOn(KeyedServiceFactory<Context>* rhs);
 protected:
  KeyedServiceFactory(const char* name, DependencyManager<Context>* manager);
  virtual ~KeyedServiceFactory();

  virtual Context* GetContextToUse(Context* context) const {
    // Safe default for Incognito mode: no service.
    if (context->IsOffTheRecord())
      return nullptr;
    return context;
  }

  // And so on...
 private:
  DependencyManager<Context*> dependency_manager_;
};

class BrowserContextDependencyManager
    : public DependencyManager<content::BrowserContext> {
};

class BrowserContextKeyedServiceFactory
    : public KeyedServiceFactory<content::BrowserContext> {
}

class BrowserStateDependencyManager
    : public DependencyManager<ios::BrowserState> {
};

class BrowserStateKeyedServiceFactory
    : public KeyedServiceFactory<ios::BrowserState> {
}
```

This solution is the safest since it does not require any new cast of the context from. It requires renaming some virtual methods but this can be dealt by `tools/git/mffr.py`.

It prevents having dependencies between `BrowserContextKeyedServiceFactory` and `BrowserStateKeyedServiceFactory` as the two types are different and don't share a base type.