Adding Page Helpers to content/

bokan@chromium.org - July 5, 2021

tl;dr

This document is a proposal and request for comment for new “Page Helper” concept to make feature development in a post-MPArch world more intuitive and less error prone. Page helpers are enabled by two new mechanisms:

- A NavigationHandleOrPageUserData that enables user data objects that are created and stored in a page-creating NavigationHandle and transferred automatically to the new Page when it is created.
- A PageRoutingWebContentsObserver that observes a WebContents and, for each observer method, calls the analogous method on the PageScopedObserver that corresponds to the page in which the state change occurred.

MPArch Background

For full details see go/mparch

Briefly, MPArch is a project to enable multiple pages being hosted in a single WebContents. This architecture will enable features such as BFCache, Prerendering, Portals and others.

The main idea is that WebContents can now host more than one frame tree. A WebContents always has a *primary* frame tree, which is the existing, familiar one that the user can see and interact with. In addition, the WebContents may also be host to one or more *non-primary* frame trees. For example: a primary page may initiate a prerender which will create a new non-primary frame tree hosted in the same WebContents. Neither the user nor the primary page can see or interact with the prerendering page in any way - it should have no effects on the browser UI.

A non-primary page may eventually be activated into the primary frame tree. For example, when a user navigates through history to a page that’s available in the BFCache, the cached page is activated into the primary frame tree, rather than loading the history entry anew. This makes the page visible and interactive to the user instantly.

Tab Helpers

`content/` has an existing concept of “Tab Helpers”. See docs/tab_helpers.md for more details. A tab helper is an object that’s both a WebContentsObserver and is owned by it.

Prior to MPArch, a WebContents hosted a single FrameTree (i.e. main frame and all its subframes). This meant tab helpers could assume to be scoped to a single frame tree. With MPArch, a WebContents is host to multiple frame tries: a “primary” one as well as zero or more non-primary (e.g. for the BackForward Cache, prerendering pages, etc.).

MPArch is a problem for existing Tab Helpers since a single instance will now observe events caused by multiple pages (e.g. a tab helper ostensibly monitoring the primary page will have DidStartNavigation called due to a navigation in a prerendering page). Tab helpers commonly make assumptions about observing a single page; e.g. resetting their state when a main frame navigates cross-document.

Rewriting individual tab helper logic to correctly account for multiple pages will require a lot of work and introduce a lot of repeated complexity in each one. Additionally, future features will need to repeat this work and understand the complex nuances of how to manually scope events and data to the correct page.

Page Helpers

I propose introducing an analogous “Page Helper” concept to `content/` to make it trivial(ish) to convert existing tab helpers to per-page semantics (I expect many tab helpers are semantically per-page) and make it so that feature authors can remain mostly ignorant of the details of how pages are hosted in a WebContents.

This requires two new mechanisms: one for storage of the helper and one for scoping observation to a single page.

1 - NavigationHandleOrPageUserData

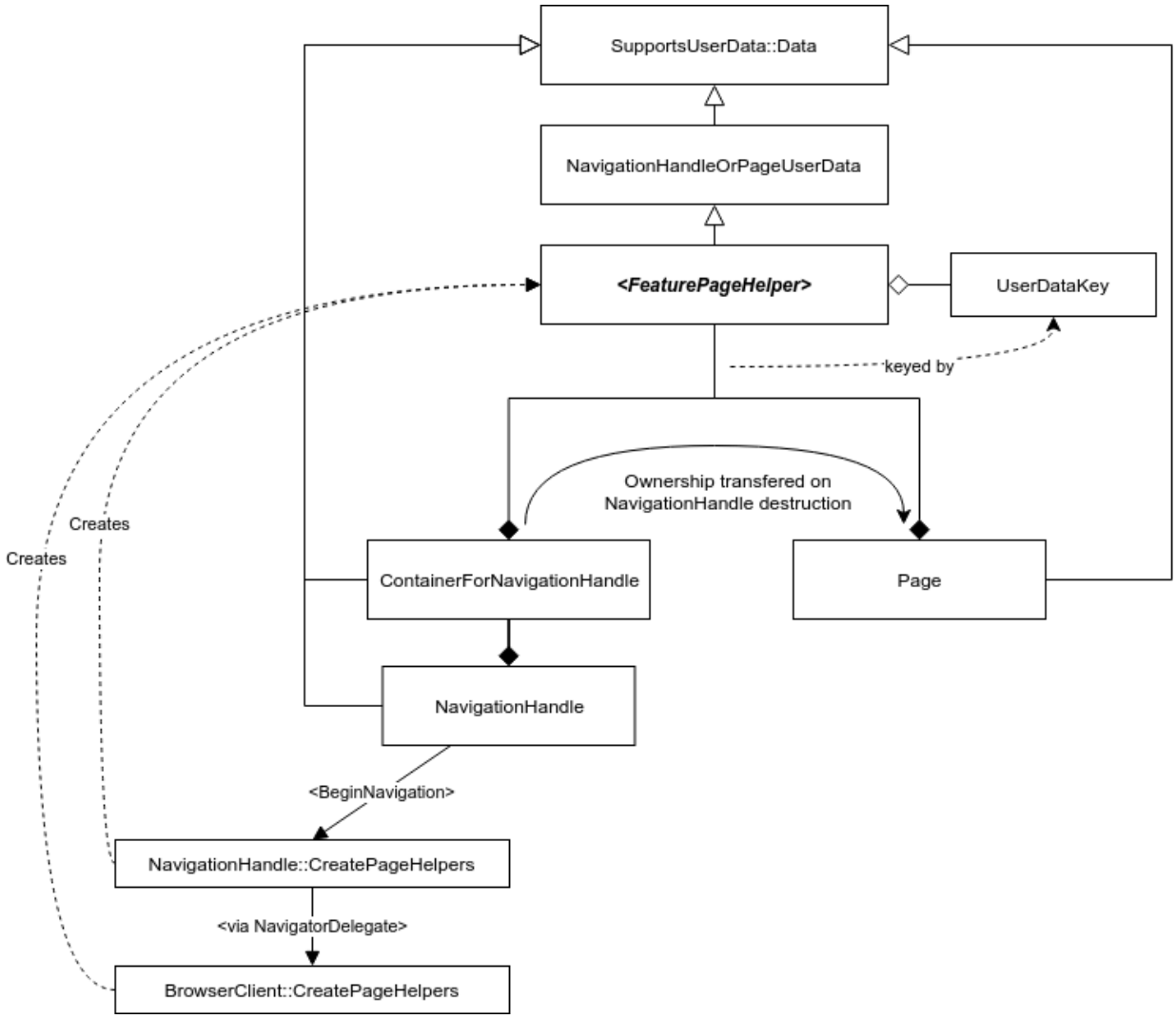
Unlike `WebContents`, `Page` is created as part of a navigation. However, it’s not uncommon for existing tab helpers to observe navigation events before a `Page` is created. This means a page helper needs to be created and stored before the `Page` is available.

Both `NavigationHandle` and `Page` derive from `base::SupportsUserData`. I propose to introduce a new helper class: `NavigationHandleOrPageUserData` (NHOPUD). A NHOPUD object is created and stored on each `NavigationHandle` that will create a new page (cross-document, main-frame navigations). If the navigation succeeds and a new `Page` is created, the NHOPUD object is automatically transferred to `Page`.

This class implements two nuanced pieces of functionality that encapsulate the complexity of MPArch away from feature authors:

1. Automatic transfer of ownership from `NavigationHandle` to `Page` when `NavigationHandle` is destroyed

2. A “smart” `FromNavigationHandle` static method that knows how to find the correct NHOPUD.



How and when is a NHOPUD created?
Page helpers will usually require a `WebContentsObserver` (see `PageRoutingWebContentsObserver`) so they could create themselves in `DidStartRequest`; however, it’d be better if NHOPUD isn’t coupled to `WebContentsObserver`.

Instead, I propose a mechanism similar to `NavigationThrottle` creation: a `CreatePageHelpers` method in `WebContentsImpl` and `BrowserClient`, allowing embedders to call a constructor from this method. This allows internal `content/` code to correctly implement the call to this method only when the NavigationHandle will lead to a new `Page` being created. It has the additional benefit that PageHelpers can be easily enumerated/audited in one place.

How and when is NHOPUD transferred to Page?
A new `Page` is created as part of a main frame's `DidCommitNavigation` on a cross-document navigation; however, we aren't observing navigation events from NHOPUD and external code can't differentiate a NHOPUD from other `NavigationHandleUserData` (and `UserData` can't be enumerated anyway).

We wrap the NHOPUD in an intermediate `ContainerForNavigationHandle` and set it as `NavigationHandleUserData`. We then use its destructor to perform the transfer (when `NavigationHandle` is destroyed, as part of `RenderFrameHostImpl::DidCommitNavigationInternal`, specifically in `Navigator::DidNavigate`). To prevent transferring onto the wrong `Page` object, the `Page` constructor can set a pointer to itself on the creating `NavigationRequest`.

Note: This requires `NavigationRequest` to manually clear its `UserData` in its destructor (deleting `ContainerForNavigationHandle`) so that a reference to `NavigationHandle` is still valid when we perform the transfer.

Why does FromNavigationHandle need to be "smart" (i.e. why not simply GetUserData from the given NavigationHandle)?
In general, finding the correct NHOPUD to use can be tricky. Using `GetUserData` on a given `NavigationHandle` will be correct only if it will create a new Page, otherwise we should be requesting it from an existing `Page`.

Since we only attach a new NHOPUD to a `NavigationHandle` when a new `Page` will be created, client code *could* try `NavigationHandle` then fallback to `Page`:

```
auto* nhopud = handle.GetUserData(NavigationHandleOrPageUserData::kUserDataKey);
if (!nhopud)
  nhopud = page.GetUserData(NavigationHandleOrPageUserData::kUserDataKey);
```

This is straightforward for same-document and subframe navigations but finding the correct page for an activating navigation is more subtle (see next question); in that case we need to get the NHOPUD from the page being activated, which is not the page whose frame tree the navigation is occurring in.

Instead of relying on client code understanding these nuances and littering code with checks like this, `NavigationHandleOrPageUserData::FromNavigationHandle` wraps this complexity for authors, getting the object from the correct Page in cases where it should.

What do we return for NHOPUD::FromNavigationHandle when the handle is for a page activation (prerender activation or BFCache restore)?
A page activation doesn't create a new `Page` so we must return an existing NHOPUD from its `NavigationHandle`. The navigation is occurring in the *primary* frame tree; however, the

activation will promote the *non-primary* page into the primary frame tree (and *implicitly* dispose of the current primary page) so we should return the NHOPUD from the non-primary page.

Intuitively, a page activation is associated with the prerendered/BFCached page, rather than the primary page. Analogously, an ordinary navigation that creates a new page would be associated with the new page, even though the old page may have initiated the navigation.

What about navigation that don't commit (e.g. 204/205 responses, a Download or "stay on current page" response)?
In this case, we only find out at response time that the navigation won't create a new page so the NHOPUD will be created and destroyed with the NavigationHandle and we won't affect state in the existing page.

IMHO this is intuitive and is likely the correct behavior in the majority of cases. Many features already have special cases in their observer methods for `!HasCommitted()` [1] [2] to avoid their usual behavior. These special-cases would either remain or can be removed if they record/affect state only after `DidFinishNavigation`.

In the (hopefully rare) cases where a non-committing navigation must affect state in the existing page, this is still possible to express by having the non-committing NHOPUD mutate state on the current `Page`'s NHOPUD.

Does this mean we'll create a new NHOPUD on each cross-document main frame navigation?
Yes, this is a difference from today where a WebContents-associated tab helper is long lived for the life of a tab. This means more construction/destruction overhead.

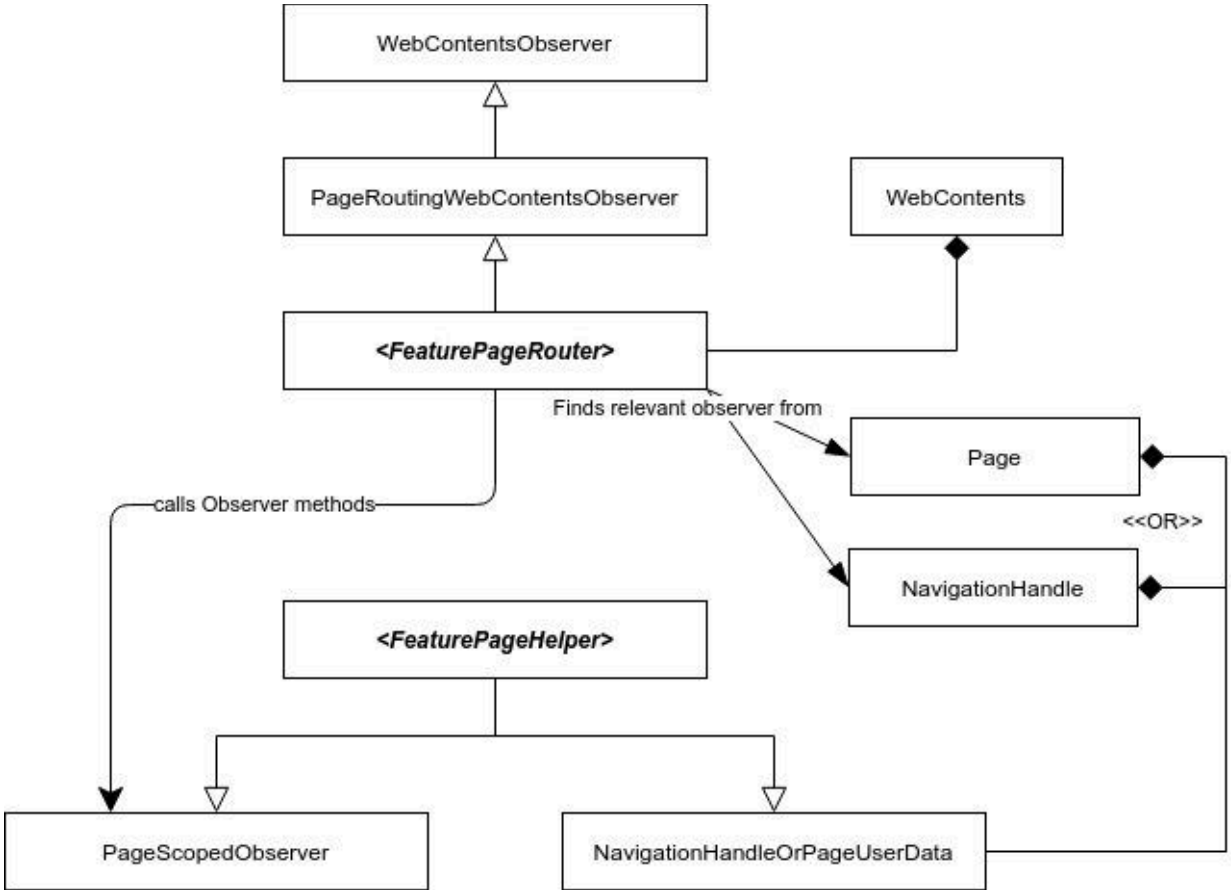
However, on the upside, many of these features are semantically per-page; they currently have complexity related to resetting state on new page creation. This change would simplify these classes and make them more natural to work with and easier to understand.

What key do we use for UserData?
A class deriving from NHOPUD declares a single user data key whose value we use in both `NavigationHandle` and `Page`.

2 - PageRoutingWebContentsObserver

With NHOPUD, a feature object is now per-page. However, it still needs some way to observe events occurring only in that page, without having to manually filter out events from unrelated pages.

I propose introducing a `PageRoutingWebContentsObserver` and `PageScopedObserver`.



A feature wishing to observe events on a single `Page` would derive from both `NavigationHandleOrPageUserData` and `PageScopedObserver`. It would then create a tab helper (i.e. `WebContentsObserver` and `WebContentsUserData`) that derives from `PageRoutingWebContentsObserver`.

`PageRoutingWebContentsObserver` observes events from all pages in the `WebContents`. For each event, it finds the relevant `PageScopedObserver` for that event and calls the analogous method on it.

In most cases, this will use `NHOPUD::FromNavigationHandle` or `NHOPUD::FromPage` on the parameters passed to the `WebContentsObserver` method, cast the result to `<FeaturePageHelper>` and call the analogous `PageScopedObserver` method.

In some cases the appropriate object can/must be derived in other ways. For example, `VisibilityChanged` or `DidGetUserInteraction` don't provide a parameter associated with a page or navigation. However, they should always be associated with the primary page's observer so in this case we can get the `PageScopedObserver` from `WebContents::GetPrimaryPage`.

`PageRoutingWebContentsObserver` can provide an intuitive, usually correct, default implementation. Most features need only derive from `PageRoutingWebContentsObserver` for the appropriate type information (to extract the correct NHOPUD) and create+attach it as a tab helper; no implementation work is required. However, if a feature has unusual semantics (for example, prerendering pages want to know about user gestures from the primary page), it could override the default behavior in `PageRoutingWebContentsObserver`. (Alternatively, if we decide this is likely to be misused or lead to confusion we can forbid it by making methods in `PageRoutingWebContentsObserver final`)

Does this mean PageScopedObserver duplicates WebContentsObserver?
Approximately, yes; any existing `WebContentsObserver` method that we'd need on `PageScopedObserver` would have to be redeclared in `PageScopedObserver` (and routing implemented in `PageRoutingWebContentsObserver`). We could do this only as needed so that `PageScopedObserver` would be a subset of the full `WebContentsObserver`.

`PageScopedObserver` *can* add new interface methods that match better to being per-page, for example, adding a `DidBecomePrimaryPage()` being called on the appropriate `PageScopedObserver` from `WebContentsObserver::DidChangePrimaryPage()`.

Additionally, if we ever want to add new observations that don't go through `WebContentsObserver`, `PageScopedObserver` alone could add those methods and register itself as an observer of its associated `Page`.

Does this mean each feature object must derive from both PageScopedObserver and NHOPUD?
Yes, this avoids adding another layer of observer registrations and iteration.

In that case, should PageScopedObserver derive from NHOPUD?
Potentially. In the above proposal there's no reason to have a PageScopedObserver that isn't a NHOPUD so this would be natural.

However, long term it might be good to have more direct observers of a `content::Page` (i.e. calling `Page::AddObserver` and having `Page` call into its observers directly). This could be done with a separate interface but having PageScopedObserver be separate from NHOPUD would allow PageScopedObserver to fill both roles.

Can/should we hide navigation events for the page activation navigation?
Page activation (e.g. restoring a page from BFCache) is a navigation in the primary main frame. This navigation is special in a number of ways: it doesn't hit the network, is synchronous once started, and doesn't create a new `RenderFrameHost` or `Page`. However, like most navigations it has an associated `NavigationHandle` and `WebContentsObservers` will see all the typical events a regular navigation will go through (e.g. `DidStartNavigation`, `ReadyToCommitNavigation`, etc.).

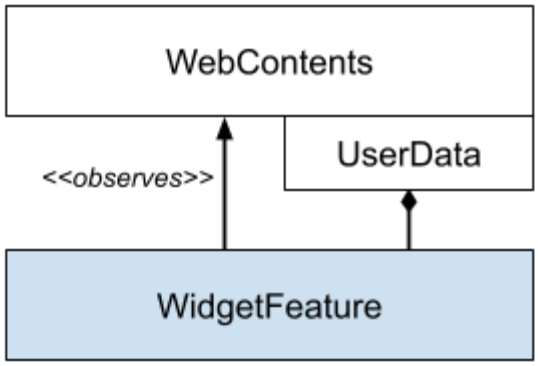
In most (all?) cases, features won't want to perform their usual functionality for a page activation and, without knowing the nuances of features like prerender and BFCache, it may be surprising that activation causes a navigation in the main frame (which doesn't initialize or destroy the page).

`PageRoutingWebContentsObserver` can avoid surprise by suppressing navigation-related events coming from a page activation. Features can (and should) watch for `DidBecomePrimaryPage` to update state when an activation does occur.

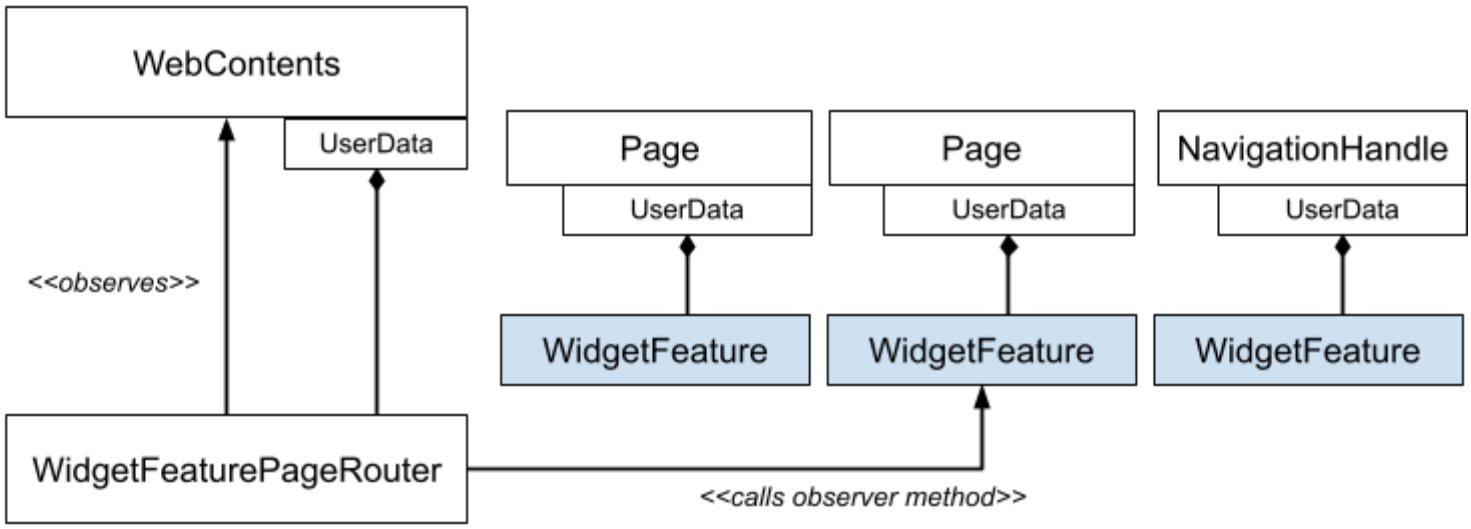
Worked Conversion Example - Subresource Filtering

Here's a generic and high level view of how TabHelpers would be converted to PageHelpers:

Before (TabHelper):



After (PageHelper):



I've prototyped the PageHelper concept by converting `ContentSubresourceFilterThrottleManager` from TabHelper to per-page. It serves as a useful example of what a conversion like this looks like for a complex feature.

The new mechanisms allow updating `ContentSubresourceFilterThrottleManager` to be per-page with minimal changes to its logic. The steps involved in the conversion:

- Introduce a new class `ContentSubresourceFilterPageRouter` that is a `PageRoutingWebContentsObserver`. This class need only implement a basic construction
    - Instantiate this class in `TabHelpers::AttachTabHelpers`
- Change `ContentSubresourceFilterThrottleManager`'s base classes from:

```
base::SupportsUserData::Data,
WebContentsObserver
```

  To:

```
NavigationHandleOrPageUserData<ContentSubresourceFilterThrottleManager>,
PageScopedWebContentsObserver
```

- Change `CreateForWebContents` to `CreateForNewPage`
    - Call `CreateForNewPage` from `ChromeContentBrowserClient::CreatePageHelpers`
- Update existing call sites of `FromWebContents` to either `FromPage` or `FromNavigationHandle`
- Update any other per-WebContents state to per-Page. Since most state is populated from observer events, this will often be a no-op. In the case of `ContentSubresourceFilterThrottleManager`, this is:
    - Replace its use of `WebContentsReceiverSet` to a page-scoped `ReceiverSet`.
    - Changing `web_contents()->GetMainFrame()` calls to `GetPage()->GetMainDocument()`

All these changes are straightforward and mechanical; they don't involve changing the "business logic" in the class.

From here, some small business logic adjustments need to be made to make the class work correctly for non-primary pages:

- `MaybeShowNotification` is called from the renderer to show a notification icon in the omnibox when a resource is blocked. A non-primary page should not affect UI so we avoid updating the UI `if (!GetPage()->IsPrimary())`
- Override `DidBecomePrimaryPage` to make the UI affecting call if `MaybeShowNotification` was called while the page was non-primary.