

Using modifyClass to change core behavior



- [When to use modifyClass](#)
 - [Basic Usage](#)
 - [Upgrading Legacy Syntax](#)
 - [Troubleshooting](#)
 - [Class already initialized](#)
-

For advanced themes and plugins, Discourse offers the `modifyClass` system. This allows you to extend and override functionality in many of core's javascript classes.

When to use `modifyClass`

`modifyClass` should be a last resort, when your customization cannot be made via Discourse's more stable customization APIs (e.g. plugin-api methods, plugin outlets, transformers).

Core's code can change at any time. And therefore, customizations made via `modifyClass` could break at any time. When using this API, you should ensure that you have controls in place to catch those issues before they reach a production site. For example, you could add automated tests to the theme/plugin, or you could use a staging site to test incoming Discourse updates against your theme/plugin.

Basic Usage

`api.modifyClass` can be used to modify the functions and properties of any class which is accessible via the Ember resolver. That includes Discourse's routes, controllers, services and components.

`modifyClass` takes two arguments:

- `resolverName` (string) - construct this by using the type (e.g. component/controller/etc.), followed by a colon, followed by the (dasherized) filename name of the class. For example: `component:d-button`, `component:modal/login`, `controller:user`, `route:application`, etc.
- `callback` (function) - a function which receives the existing class definition, and then returns an extended version.

For example, to modify the `click()` action on d-button:

```
api.modifyClass(
  "component:d-button",
  (Superclass) =>
    class extends Superclass {
      @action
      click() {
        console.log("button was clicked");
        super.click();
      }
    }
);
```

The `class extends ...` syntax mimics that of [JS child classes](#). In general, any syntax/features supported by child classes can be applied here. That includes `super`, static properties/functions, and more.

However, there are some limitations. The `modifyClass` system only detects changes to the class's JS `prototype`. Practically, that means:

- introducing or modifying a `constructor()` is not supported

```
api.modifyClass(
  "component:foo",
  (Superclass) =>
    class extends Superclass {
      constructor() {
        // This is not supported. The constructor will be ignored
      }
    }
);
```

- introducing or modifying class fields is not supported (although some decorated class fields, like `@tracked` can be used)

```
api.modifyClass(
  "component:foo",
  (Superclass) =>
    class extends Superclass {
      someField = "foo"; // NOT SUPPORTED - do not copy
      @tracked someOtherField = "foo"; // This is ok
    }
);
```

- simple class fields on the original implementation cannot be overridden in any way (although, as above, `@tracked` fields can be overridden by another `@tracked` field)

```
// Core code:
class Foo extends Component {
  // This core field cannot be overridden
  someField = "original";

  // This core tracked field can be overridden by including
  // `@tracked someTrackedField =` in the modifyClass call
  @tracked someTrackedField = "original";
}
```

If you find yourself wanting to do these things, then your use-case may be better satisfied by making a PR to introduce new APIs in core (e.g. plugin outlets, transformers, or bespoke APIs).

Upgrading Legacy Syntax

In the past, `modifyClass` was called using an object-literal syntax like this:

```
// Outdated syntax - do not use
api.modifyClass("component:some-component", {
  someFunction() {
    const original = this._super();
    return original + " some change";
  }
  pluginId: "some-unique-id"
});
```

This syntax is no longer recommended, and has known bugs (e.g. overriding getters or `@actions`). Any code using this syntax should be updated to use the native-class syntax described above. In general, conversion can be done by:

1. removing `pluginId` - this is no longer required
2. Update to the modern native-class syntax described above
3. Test your changes

Troubleshooting

Class already initialized

When using `modifyClass` in an initializer, you may see this warning in the console:

```
“ Attempted to modify "{name}", but it was already initialized earlier in the  
boot process
```

In theme/plugin development, there are two ways this error is normally introduced:

- **Adding a `lookup()` caused the error**

If you `lookup()` a singleton too early in the boot process, it will cause any later `modifyClass` calls to fail. In this situation, you should try to move the lookup to happen later. For example, you would change something like this:

```
// Lookup service in initializer, then use it at runtime (bad!)  
export default apiInitializer((api) => {  
  const composerService = api.container.lookup("service:composer");  
  api.composerBeforeSave(async () => {  
    composerService.doSomething();  
  });  
});
```

To this:

```
// 'Just in time' lookup of service (good!)  
export default apiInitializer((api) => {  
  api.composerBeforeSave(async () => {  
    const composerService = api.container.lookup("service:composer");  
    composerService.doSomething();  
  });  
});
```

```
});
```

- **Adding a new `modifyClass` caused the error**

If the error is introduced by your theme/plugin adding a `modifyClass` call, then you'll need to move it earlier in the boot process. This commonly happens when overriding methods on services (e.g. `topicTrackingState`), and on models which are initialized early in the app boot process (e.g. a `model:user` is initialized for `service:current-user`).

Moving the `modifyClass` call earlier in the boot process normally means moving the call to a `pre-initializer`, and configuring it to run before Discourse's 'inject-discourse-objects' initializer. For example:

```
// (plugin)/assets/javascripts/discourse/pre-initializers/extend-user-for-my-
plugin.js
// or
// (theme)/javascripts/discourse/pre-initializers/extend-user-for-my-plugin.js

import { withPluginApi } from "discourse/lib/plugin-api";

export default {
  name: "extend-user-for-my-plugin",
  before: "inject-discourse-objects",

  initializeWithApi(api) {
    api.modifyClass("model:user", (Superclass) => class extends Superclass {
      myNewUserFunction() {
        return "hello world";
      },
    });
  },

  initialize() {
    withPluginApi(this.initializeWithApi);
  },
};
```

This modification of the user model should now work without printing a warning, and the new method will be available on the `currentUser` object.

🔖 #2

📅 2026-10-31 13:28:11 UTC

📅 2026-10-31 13:28:11 UTC