# Using Transformers to customize client-side values and behavior


- Value Transformers
  - registerValueTransformer => boolean
- Behavior Transformers
  - registerBehaviorTransformer => boolean
- Multiple registered transformers
- Finding Transformers
- Introducing new Transformers
- TypeScript


Discourse core includes a number of "transformer" hooks which can be used to customize client behavior. These fall into two categories:

- **Value Transformers** take an output from Discourse core, and optionally modify it, or replace it with something else
- **Behavior Transformers** can replace a particular piece of logic in Discourse core, and optionally call the original implementation.

Each transformer hook is referenced by a name, and multiple plugins/themes can register transformers against the same hook.

# Value Transformers

Value transformers can be registered using `api.registerValueTransformer`:

### `registerValueTransformer` => `boolean`

| Param | Type | Description |
| --- | --- | --- |
| `transformerName` | `string` | the name of the transformer |
| `valueCallback` | `function({value, context})` | callback to be used to transform the value |

The `valueCallback` will be passed two named arguments: the original value, and a context object. It should return the transformed value. Or, if no transformation is required, it should return the original value.

Callbacks can access the provided context for the current transformer hook, or they can reference other data sources. Many value transformers are executed in autotracking contexts, which means that referencing reactive state will cause the transformer to be automatically re-evaluated when that state changes.

For example, to modify the link URL of the logo on small screen devices:

```
api.registerValueTransformer("home-logo-href", ({ value, context }) => {
  const capabilities = api.container.lookup("service:capabilities");
  if (!capabilities.viewport.sm) {
    // viewport narrower than the `sm` breakpoint
    return "/latest";
  } else {
    return value;
  }
});
```

# Behavior Transformers

Behavior transformers are similar, but instead of receiving and returning a value, they wrap some original behavior, and can optionally call the original implementation.

### `registerBehaviorTransformer` => `boolean`

| Param | Type | Description |
| --- | --- | --- |
| `transformerName` | `string` | the name of the transformer |
| `behaviorCallback` | `function({next, context})` | callback to be used to transform or override the behavior |

The `behaviorCallback` is passed two named arguments. `next` is a function which can optionally be called to run the original behavior. `context` contains extra information which may be useful.

For example, if you wanted to limit infinite loading to 100 topics:

```
api.registerBehaviorTransformer(
  "discovery-topic-list-load-more",
  ({ next, context }) => {
    const topicList = context.model;
    if (topicList.topics.length > 100) {
      alert("Not loading any more");
    } else {
      next();
    }
  }
);
```

# Multiple registered transformers

If multiple transformers are registered against a single name, then they will be run in order of registration. The input to valueTransformers will be the value returned by the previous transformer. The `next()` function passed to behaviorTransformers will call the remaining transformers in the chain.

# Finding Transformers

At the moment there is no centralized documentation for transformers. However, a list of all core transformers can be found in the code here. You can then search for those names in the core codebase to find where they're called, the context/value, and the expected return type.

# Introducing new Transformers

To introduce new transformers, the name first needs to be added to the registry. In core, add it to lib/registry/transformers.ts.

To introduce a new transformer via a theme or plugin, create a `pre-initializer` and register the name via the relevant API:

```
// .../discourse/pre-initializers/my-transformer.js
export default {
  before: "freeze-valid-transformers",
  initialize() {
    withPluginApi((api) => {
      api.addValueTransformerName("my-value-transform");
      api.addBehaviorTransformerName("my-behavior-transform");
    });
  },
};
```

To apply the transformer to some value or behavior, import the associated `apply` function and use it anywhere in your code:

```
import {
  applyValueTransformer,
  applyBehaviorTransformer,
} from "discourse/lib/transformer";

const context = { bestForumSoftware: "Discourse" };

// Value Transformer
const originalValue = "Hello";
const transformedValue = applyValueTransformer(
  "my-value-transform",
  originalValue,
```

```
  context
);

// Behavior Transformer
applyBehaviorTransformer(
  "my-behavior-transformer",
  () => {
    alert("This is the core implementation");
  },
  context
);
```

# TypeScript

The transformer pipeline exports `ValueTransformer`, `MutableValueTransformer`, `AnyValueTransformer`, `BehaviorTransformer`, and `DefaultBehaviorTransformer` callback types from `discourse/lib/transformer`. Value callbacks take value and context type parameters: `ValueTransformer<T, C>` returns `T`, while `MutableValueTransformer<T, C>` mutates the value and returns `void`. `registerValueTransformer` accepts `AnyValueTransformer<T, C>`, which allows either kind but still rejects a callback returning something other than `T`. Behavior callbacks take return and context type parameters. Context types can be interfaces without an index signature. Hook names remain strings so plugins and themes can introduce hooks.

`applyValueTransformer` and `applyMutableValueTransformer` infer their result type from the default value. Supply an explicit type argument when a hook can return a broader type than its default, for example `applyValueTransformer<string | null>("custom-label", null)`. Registration and application must agree on a hook's value, context, and return types; the string registry does not enforce a shared contract between separate calls.

Behavior callbacks receive a shallow copy of the context. `next()` and `applyBehaviorTransformer` can return `void` when middleware stops the chain or calls `next()` after the queue is exhausted. Promise results pass through unchanged, and async middleware that awaits `next()` resolves to the awaited result or `undefined`; `BehaviorTransformerResult<R>` describes the full result type.

---

□□□ #2
□ □□□ □□□ 2026-10-31 13:28:35 UTC
□ □□□ □□□ 2026-10-31 13:28:36 UTC