The Bombadil Manual

Version 0.3.1

Contents

Introduction

Bombadil is property-based testing for web UIs, autonomously exploring and validating correctness properties, *finding harder bugs earlier*. It runs in your local developer environment, in CI, and inside Antithesis.

Why Bombadil?

Or rather, *why property-based testing?* Because example-based testing, especially in the area of browser testing, is costly and limited:

- Costly, because maintaining suites of Playwright or Cypress tests takes a lot of work. And even in the days of AI agents writing and updating those tests for you, they easily break and require your attention.
- Limited, in that they only test very small parts of the state space. A bunch of happy cases, a set of regression tests, and maybe even some error handling cases that are important. But what about everything else? All the stuff you or the agent didn't think about testing?

This is where property-based testing, or *fuzzing* if you will, comes into play. By randomly and systematically searching the state space, Bombadil behaves in ways you didn't think about testing for. Unexpected sequences of actions, weird timings, strange inputs that you forgot could be entered.

How it works

Instead of describing "what good looks like" in terms of fixed test cases, you express general properties of your system, how it should behave in all cases. Bombadil checks each property as it explores your system in its chaotic ways, reporting back any violations.

To test a web application using Bombadil, you write a specification in TypeScript that exports properties and action generators. These can be domain-specific — to exercise and validate your system's logic in custom ways — or be imported from the defaults provided by Bombadil. It doesn't matter how the application is built — if it's a single-page app, server-side rendered, or even static HTML — Bombadil tests anything that uses the DOM.

Conceptually, it runs in a loop doing the following:

1. Extracts the current state from the browser
2. Checks all properties against the current state, recording violations[1]
3. Selects the next action based on the current state, and performs it
4. Waits for the next event (page navigation, DOM mutation, or timeout)
5. *Goes to step 1*

Bombadil itself decides what is an interesting event and when to capture state. The specification author provides the properties and actions, Bombadil does the rest.

[1]You can also configure Bombadil to exit on the first found violation.

Getting started

Bombadil runs on your development machine if you're on macOS or Linux. You can use it to validate changes to TypeScript specifications, and to run short tests while working on your system. Then you'll have something like GitHub Actions to run longer tests on your main branch or in nightlies.

Installation

The most straightforward way for you to get started is downloading the executable for your platform:

macOS

Download the `bombadil` binary using `curl` (or `wget`) and make it executable:

```
curl -L -o bombadil https://github.com/antithesishq/bombadil/releases/
    ↪ download/v0.3.1/bombadil-aarch64-darwin
chmod +x bombadil
```

Put the binary somewhere on your `PATH`, like in `~/.local/bin` if that is configured.

```
mv ./bombadil ~/.local/bin/bombadil
```

You should now be able to run it:

```
bombadil --version
```

WARNING

Do not download the executable with your web browser. It will be blocked by GateKeeper.

Linux

Download the `bombadil` binary and make it executable:

```
curl -L -o bombadil https://github.com/antithesishq/bombadil/releases/
    ↪ download/v0.3.1/bombadil-x86_64-linux
chmod +x bombadil
```

Put the binary somewhere on your `PATH`, like in `~/.local/bin` if that is configured.

```
mv ./bombadil ~/.local/bin/bombadil
```

You should now be able to run it:

```
bombadil --version
```

Nix (flake)

```
nix run github:antithesishq/bombadil
```

Not yet available, but coming soon:

- executables bundled in NPM package (i.e. `npx @antithesishq/bombadil`)
- Docker images
- a GitHub Action, ready to be used in your CI configuration

If you want to compile from source, see Contributing.

TypeScript support

When writing specifications in TypeScript, you'll want the types available. Get them from NPM with your package manager of choice:

npm

```
npm install --save-dev @antithesishq/bombadil
```

Yarn

```
yarn add --dev @antithesishq/bombadil
```

Bun

```
bun add --development @antithesishq/bombadil
```

Or use the files provided in the release package.

Your first test

With the CLI installed, let's run a test just to see that things are working:

```
bombadil test https://en.wikipedia.org --output-path my-test
```

This will run until you shut it down using CTRL+C. Any property violations will be logged as errors, and with the `--output-path` option you get a JSONL file to inspect afterwards.

Find the URLs with violations (assuming you have `jq` installed):

```
jq -r 'select(.violations != []) | .url' my-test/trace.jsonl
```

Nothing? That's fine, Wikipedia is pretty solid! This confirms that Bombadil runs and produces results.

NOTE

Bombadil doesn't yet produce a human-readable test report, so this requires some `jq` trickery. Stay tuned, better UIs are on their way!

Specification language

To extend Bombadil with domain-specific knowledge, you write specifications. These are plain TypeScript or JavaScript modules using the library provided by Bombadil, exporting *properties* and *action generators*.

Here's how you run Bombadil with a custom specification:

```
bombadil test https://example.com example.ts
```

For a full listing of CLI options, see the reference.

Structure

A specification is a regular ES module. The following examples use TypeScript, but you may also write them in JavaScript.

NOTE

If you do use TypeScript, you'll want to install the types from @antithesishq/bombadil.

Both properties and action generators are exposed to Bombadil as exports:

```
export const myProperty = ...;

export const myAction = ...;
```

You may split up your specification into multiple modules and structure it the way you like, but the top-level specification you give to Bombadil must only export properties and action generators.

Default properties and action generators

Bombadil comes with a set of default properties and action generators that work for most web applications. You'll probably want to reexport all or at least most of these:

```
export * from "@antithesishq/bombadil/defaults";
```

In fact, this is exactly what is used when running tests without a custom specification file. If you want to selectively pick just a subset of these, use the following modules:

```
export {
    noUncaughtExceptions
} from "@antithesishq/bombadil/defaults/properties";
export {
    clicks,
    reload,
} from "@antithesishq/bombadil/defaults/actions";
```

The defaults include properties checking for uncaught exceptions, unhandled promise rejections, error logs, HTTP 4xx and 5xx responses, and more. On the actions side, there are generators for general navigation and interaction with semantic HTML elements.

You may freely combine defaults with your own properties and action generators.

Language features

The specification language of Bombadil, embedded in TypeScript or JavaScript, has a small set of central concepts. This section describes them in detail.

Properties

A property is a description of how the system under test should behave *in general*. This is different from example-based testing (Playwright, Cypress) where you describe how it behaves for *particular* cases.

The most intuitive kind of property, which you might have come across before, is an *invariant*: a condition that should always be true. In Bombadil, invariants are expressed using the `always` temporal operator:

```
always(
    // some condition that should always be true
)
```

To instruct Bombadil to check your property, you must export it from your specification module. Its name is used in error reports, so give the export a meaningful name.

```
export const pageHasTitle = always(
    // check that there's a page title somehow
);
```

You may export multiple properties, including the defaults, and they'll all be checked independently. But how do you "check that there's a page title somehow"? You need access to the browser, and for that, you use *extractors*.

Extractors

In order to describe a condition about the web page you're testing, you first need to extract state. This is done with the `extract` function, which runs inside the browser on every state that Bombadil decides to capture.

```
extract(state => ...)
```

You give it a function that takes the current browser state as an argument, and returns JSON-serializable data. The state object contains a bunch of things, but most important are `document` and `window`, the same ones you have access to in JavaScript running in a browser.

To extract the page title, you'd define this at the top level of your specification:

```
const pageTitle = extract(state => state.document.title || "");
```

The `pageTitle` value is not a `string` though — it's a `Cell<string>`, a stateful value that changes over time. For every new state captured by Bombadil, the extractor function gets run, and the cell is updated with its return value.

Using the `pageTitle` cell, you can define the property:

```
export const pageHasTitle = always(() =>
    pageTitle.current !== ""
);
```

Two things to note about this example:

1. The expression passed to `always` is a function that takes no arguments — a *thunk*. This is because it needs to be evaluated in every state. It needs to *always* be true, not just once, and that's why you need to supply the thunk rather than a `boolean`.
2. To get the `string` value out of the cell, you use `pageTitle.current`.

This is a custom property using the *temporal* operator called `always`. There are other temporal operators, described in Formulas.

Formulas

Formulas and temporal operators may sound scary, but fear not — they are essentially ways of expressing "conditions over time". Here are some quick facts about formulas and temporal operators:

- Temporal operators return formulas.
- Every property in Bombadil is a formula (of the `Formula` type).
- A temporal operator is a function that takes some subformula and evaluates it over time.
- Different temporal operators evaluate their subformulas in different ways.
- Bombadil evaluates formulas against a sequence of states to check if they *hold true*.

In addition to `always`, there's also `eventually` and `next`. Here's an informal[2] description of how they work:

- `always(x)` holds if `x` holds in *this* and *every future* state
- `next(x)` holds if `x` holds in *the next* state
- `eventually(x)` holds if `x` holds in *this* or *any future* state

They accept *subformulas* as arguments, but in the example with `always` above, the argument was a thunk. This works because the operators automatically convert thunks into formulas. There's an operator for doing that explicitly, called `now`:

```
always(now(() => pageTitle.current !== ""))
```

You normally don't have to use the `now` operator, unless you want to use *logical connectives* at the formula level. They are defined as methods on formulas:

- `x.and(y)` holds if `x` holds and `y` holds
- `x.or(y)` holds if `x` holds or `y` holds
- `x.implies(y)` holds if `x` doesn't hold or `y` holds

There's also negation, both as a function and as a method on formulas, i.e. `not(x)` and `x.not()`.

The `now` operator is useful when expressing single-state preconditions. The following property checks that pressing a button shows a spinner that is eventually hidden again:

```
const buttonPressed = extract(() => ...);
const spinnerVisible = extract(() => ...);

now(() => buttonPressed.current).implies(
```

[2] Formally, the properties in Bombadil use a flavor of Linear Temporal Logic, if you're into dense theoretical stuff.

```
    now(() => spinnerVisible.current).and(eventually(() => !spinnerVisible.current
    ↪ ))
)
```

You can build more advanced formulas, even with nested temporal operators, but the basics are often powerful enough. See the examples at the bottom for more inspiration.

Action generators

In addition to exporting properties in a specification, you export action generators. A generator is an object with a `generate()` method. An action generator is such an object that generates values of type `Tree<Action>`.

Like with default properties, there are default actions provided by Bombadil. These will get you a long way, but there are times where you need to define your own action generators.

For every state that Bombadil captures, all action generators are run, contributing to a tree structure of *possible* actions. Bombadil then randomly picks one in that tree. Why a tree, though? It's because the branches are *weighted* — by default they're equally weighted, but you can override this to control the probability of an action being picked.

To define a custom action generator, you use the `actions` function, which takes a thunk that returns an array of actions:

```
export const myAction = actions(() => {
    return [
        ...
    ];
});
```

In the returned array, each element is a value of the following `Action` type, provided by the NPM package:

```
interface Point {
    x: number;
    y: number;
}

type Action =
    | "Back"
    | "Forward"
    | "Reload"
    | { Click: { name: string; content?: string; point: Point } }
    | { TypeText: { text: string; delayMillis: number } }
    | { PressKey: { code: number } }
    | { ScrollUp: { origin: Point; distance: number } }
    | { ScrollDown: { origin: Point; distance: number } };
```

Here's a generator for clicks in the center of a `canvas` element:

```
const canvasCenter = extract((state) => {
    const canvas = state.document.querySelector("#my-canvas");
    if (!canvas) {
        return null;
    }
    const rect = canvas.getBoundingClientRect();
    if (rect.width > 0 && rect.height > 0) {
        return {
            x: rect.left + rect.width / 2,
            y: rect.top + rect.height / 2,
        };
    }
    return null;
});


export const clickCanvas = actions(() => {
    const point = canvasCenter.current;
    return point ? [{ Click: { name: "canvas", point } }] : [];
});
```

The actions you return must be possible to perform in the current state. Your action generators should therefore depend on cells and validate your actions before returning them, as done with `canvasCenter` in the previous example. Another example is the `back` action generator provided by Bombadil, which checks that there's a history entry to go back to, otherwise returning `[]`.

To give actions different weights, use the `weighted` combinator and wrap each subgenerator in an array with the weight as the first element:

```
export const navigation = weighted([
    [10, back],
    [1, forward],
    [1, reload],
]);
```

Examples

These are full, runnable examples of properties and action generators you might need in your own testing with Bombadil. Think of them as design patterns for properties. Each example is a self-contained specification file.

Invariant: max notification count

This is a simple one checking that there are never more than five notifications shown.

```
import { extract, always } from "@antithesishq/bombadil";
export * from "@antithesishq/bombadil/defaults";

const notification_count = extract((state) =>
    state.document.body.querySelectorAll(".notification").length,
);

export const max_notifications_shown = always(() =>
    notification_count.current <= 5,
);
```

Sliding window: constant notification count

This property checks that the notification count doesn't change — that it is the same as in the first state. Note how this property evaluates `time.current` in the outer thunk, and then uses that time value to look up older values.

```
import { extract, always, now, time } from "@antithesishq/bombadil";
export * from "@antithesishq/bombadil/defaults";

const notification_count = extract((state) =>
    state.document.body.querySelectorAll(".notification").length,
);

export const constantNotificationCount = now(() => {
    const start = time.current;
    return always(() =>
        notification_count.current === notification_count.at(start),
    );
});
```

Guarantee: error disappears

A *guarantee property* checks that something good eventually happens, within some time bound. Here is a property that checks that error messages disappear within five seconds.

```
import { extract, always, now, eventually } from "@antithesishq/bombadil";
export * from "@antithesishq/bombadil/defaults";

const errorMessage = extract((state) =>
    state.document.body.querySelector(".error")?.textContent ?? null,
);

export const errorDisappears = always(
    now(() => errorMessage.current !== null).implies(
        eventually(() => errorMessage.current === null)
            .within(5, "seconds"),
    ),
);
```

Contextful guarantee: notification includes past value

This example uses an outer thunk to force a cell value (`nameEntered`) at every state, and then closes over that value with the inner thunk passed to `eventually`. The property checks that if there's a non-blank name entered, and it is submitted, then eventually there will be a notification that includes the name.

```
import { extract, always, now, next, eventually } from "@antithesishq/bombadil";
export * from "@antithesishq/bombadil/defaults";

const name = extract((state) => {
    const element =
        state.document.body.querySelector("#name-field");
    return (element as HTMLInputElement | null)?.value ?? null;
});
```

```
const submitInProgress = extract((state) =>
    state.document.body.querySelector("submit.progress")
        !== null,
);

const notificationText = extract((state) =>
    state.document.body.querySelector(".notification")?.textContent
        ?? null,
);

export const notificationIncludesMessage = always(() => {
    const nameEntered = name.current?.trim() ?? "";

    return now(() => nameEntered !== "")
        .and(next(() => submitInProgress.current))
        .implies(eventually(() =>
            notificationText.current?.includes(nameEntered)
                ?? false,
        ).within(5, "seconds"));
});
```

State machine: counter

This property models a counter as a state machine, checking that the counter only transitions by staying the same, incrementing by 1, or decrementing by 1 (no invalid jumps allowed).

```
import { extract, always, now, next } from "@antithesishq/bombadil";
export * from "@antithesishq/bombadil/defaults";

const counterValue = extract((state) => {
    const element = state.document.body.querySelector("#counter");
    return parseInt(element?.textContent ?? "0", 10);
});

const unchanged = now(() => {
    const current = counterValue.current;
    return next(() => counterValue.current === current);
});

const increment = now(() => {
    const current = counterValue.current;
    return next(() => counterValue.current === current + 1);
});

const decrement = now(() => {
    const current = counterValue.current;
    return next(() => counterValue.current === current - 1);
});

export const counterStateMachine =
    always(unchanged.or(increment).or(decrement));
```

If this specification exports the `reload` action, the `unchanged` property becomes relevant[3]. Unless this application

[3]A state transition that allows for nothing to change is a way of making a property “stutter-invariant”, as it’s called in the literature.

stored the state of the counter somehow, reloading the page would clear the counter, which this property would catch as a violation.

Reference

Command-line interface

bombadil test

`bombadil test [OPTIONS] <ORIGIN> [SPECIFICATION_FILE]`

Argument	Description
`<ORIGIN>`	Starting URL of the test (also used as a boundary so that Bombadil doesn't navigate to other websites)
`[SPECIFICATION_FILE]`	A custom specification in TypeScript or JavaScript, using the `@antithesishq/bombadil` package on N

Option	Description
`--output-path <OUTPUT_PATH>`	Where to store output data (trace, screenshots, etc)
`--exit-on-violation`	Whether to exit the test when first failing property is found (useful in deve
`--width <WIDTH>`	Browser viewport width in pixels
`--height <HEIGHT>`	Browser viewport height in pixels
`--device-scale-factor <DEVICE_SCALE_FACTOR>`	Scaling factor of the browser viewport, mostly useful on high-DPI monitor
`--headless`	Whether the browser should run in a visible window or not
`--no-sandbox`	Disable Chromium sandboxing
`-h, --help`	Print help

bombadil test-external

`bombadil test-external [OPTIONS] <ORIGIN> [SPECIFICATION_FILE]`

Argument	Description
`<ORIGIN>`	Starting URL of the test (also used as a boundary so that Bombadil doesn't navigate to other websites)
`[SPECIFICATION_FILE]`	A custom specification in TypeScript or JavaScript, using the `@antithesishq/bombadil` package on N

Option	Description
`--output-path <OUTPUT_PATH>`	Where to store output data (trace, screenshots, etc)
`--exit-on-violation`	Whether to exit the test when first failing property is found (useful in deve
`--width <WIDTH>`	Browser viewport width in pixels
`--height <HEIGHT>`	Browser viewport height in pixels
`--device-scale-factor <DEVICE_SCALE_FACTOR>`	Scaling factor of the browser viewport, mostly useful on high-DPI monitor
`--remote-debugger <REMOTE_DEBUGGER>`	Address to the remote debugger's server, e.g. http://localhost:9222
`--create-target`	Whether Bombadil should create a new tab and navigate to the origin URL
`-h, --help`	Print help