The Bombadil Manual

Version 0.6.1

Contents

Introduction

Bombadil is property-based testing[1] for user interfaces. It autonomously explores and validates 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 when browser testing, is costly and limited:

- Costly, because maintaining suites of Playwright or Cypress tests takes a lot of work. Even in the age of AI, tests written and updated by coding agents can 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 — like 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, defining 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 Bombadil's defaults. Bombadil tests anything that uses the DOM, no matter how it's built. This includes single-page apps, server-side rendered apps, and even static HTML.

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[2]
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. *Returns to step 1*

Bombadil itself decides what is an interesting event and when to capture state. You provide the properties and actions, Bombadil does the rest!

[1] See the property based testing guide for an introduction.

[2] 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. Use it to validate changes to TypeScript specifications, and to run short tests while working on your system. Configure something like GitHub Actions or a cron job to continously run longer tests on your main branch or in a deployed staging environment. When you're ready to put your system under extreme conditions, and test your full stack deterministically, run it inside Antithesis.

Installation

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

npm

Install as a development dependency in your project:

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

Add a script to your `package.json` to run Bombadil:

```
{
  "scripts": {
    "test": "bombadil browser test --time-limit=1m https://your-app.
    ↪ example.com"
  }
}
```

Then run it with `npm test`. This also provides TypeScript type definitions for writing specifications.

macOS

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

```
curl -L -o bombadil https://github.com/antithesishq/bombadil/releases/
    ↪ download/v0.6.1/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/v0.6.1/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
```

GitHub Actions

```
- uses: antithesishq/bombadil-action@v2
  with:
    driver: browser
    origin: https://your-app.example.com
    specification: ./bombadil/specification.ts
    time-limit: 5m
    exit-on-violation: true
    output-path: bombadil-output
```

See its README for detailed instructions.

> **NOTE**
>
> Docker images are not yet available, but coming soon.

If you want to compile from source, see Contributing.

TypeScript support

When writing specifications in TypeScript, you'll want the types available. If you installed Bombadil via npm, you already have them — skip ahead to Your first test.

Otherwise, install the package 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 browser test just to see that things are working:

```
bombadil browser 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 will get results to inspect afterwards.

Launch the *Bombadil Inspect* tool to see what happened in the test you just ran:

```
bombadil browser inspect my-test
```

This will open a web application in your browser, which has some features to highlight:

- This interface is focused on *state transitions*, i.e. the state before and after each action.
- On the left is the actions list, which you can use to navigate the state transitions by clicking the actions.
- In the bottom you'll see the timeline, which you can scrub (click and drag) with your mouse. The timeline also shows the currently selected state transition.
- If there were any violations found in the test, they'll be shown as exclamation mark icons in the timeline.

No violations? That's fine, Wikipedia is pretty solid! This confirms that Bombadil runs and produces results.

Reproducing violations

If Bombadil finds a bug in a project you're working on, you might want to reproduce that test case when working on a bug fix. That is, have Bombadil perform the same sequence of actions to reach the same state.

Use the `--reproduce` option and point it to the output directory of the original test run:

```
bombadil browser test --reproduce=my-test http://example.com
```

Reproductions are not guaranteed to succeed; if they diverge, Bombadil fails with an error. For reproductions to succeed, it's important to use the same options as in the original test. After a test run, Bombadil prints both the `inspect` and `--reproduce` commands you can use.

Specification language

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

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

```
bombadil browser 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. 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.

Importing modules and files

You can split your specification up into multiple modules and import them from the top level specification module:

```
import { thing } from "./lib/thing.ts";
```

If you install packages from NPM or similar, you may import those too:

```
import equal from "deep-equal";
```

The runtime is limited, so many existing packages might not work, e.g. if they import NodeJS packages.

Non-code files

You can import JSON, text, or binary data. This is useful for bootstrapping tests with reference data, dictionaries, configuration, etc. Use ES import attributes to specify the type of import:

```
import data from "./fixtures/data.json" with { type: "json" };
import contents from "./wordlist.txt" with { type: "text" };
import raw from "./snapshot.dat" with { type: "binary" };
```

When using `"text"` you get a `string`, and when using `"binary"` you get `Uint8Array`.

For JSON data, you can also just use the file extension:

```
import data from "./fixtures/data.json";
```

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/browser/defaults";
```

In fact, these defaults are exactly what are used when running tests without a custom specification file. If you want to selectively pick just a subset of these, replace the `*` with the relevant names:

```
export {
    // Properties
    noUncaughtExceptions,
    // Actions
    clicks,
    reload,
} from "@antithesishq/bombadil/browser/defaults";
```

You may freely combine defaults with your own properties and action generators. All properties and action generators exported by the top-level module are used by Bombadil.

The browser 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.

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 (e.g. Playwright, Cypress, or Selenium) 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 hasTitle = always(
```

```
    // check that there's a title rendered 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 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 the most important are `document` and `window` — the same ones you have access to in JavaScript running in a browser.

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

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

The `title` 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 `title` cell, you can define the property:

```
export const hasTitle = always(() =>
    title.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 `.current`.

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

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*.

Temporal operator types include `always`, as discussed in the example in Extractors above, and also `eventually` and `next`. Here's an informal[3] 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. You'll notice in the example with `always` above, the argument was a thunk. This still works, because the operators automatically convert thunks into formulas. In fact, there's an operator for doing that explicitly, called `now`:

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

You can build more advanced formulas, and even include 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 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'll 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* — equally, by default. 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 [
        ...
```

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

```
    ];
});
```

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"
    | "Wait"
    | { Click: { name: string; content?: string; point: Point } }
    // Many others...
    ;
```

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 } }] : [];
});
```

For double-click actions, specify the delay between clicks in milliseconds (0-1000ms):

```
export const doubleClickCanvas = actions(() => {
    const point = canvasCenter.current;
    return point ? [{
        DoubleClick: {
            name: "canvas",
            point,
            delayMillis: 100,
        }
    }] : [];
```

```
});
```

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 property checking that there are never more than five notifications shown.

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

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

export const max_notifications_shown = always(() =>
    notificationCount.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 `notificationCount.current` in the outer thunk, and then uses that in the inner thunk to compare against the current value.

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

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

export const constantNotificationCount = now(() => {
    const initial = notificationCount.current;
```

```
    return always(() =>
        notificationCount.current === initial,
    );
});
```

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/browser/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 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. 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`.

```
import { extract, always, now, next, eventually } from "@antithesishq/bombadil";
export * from "@antithesishq/bombadil/browser/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/browser/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.[4] Unless this application stored the state of the counter somehow, reloading the page would clear the counter, which this property would catch as a violation.

[4] 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.

Reference

Command-line interface

Exit codes

Code	Meaning
0	Test completed normally (including time limit)
1	Other error
2	Property violation(s) detected

bombadil browser 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.)
`--output-path-overwrite`	Overwrite any existing `trace.jsonl` at `--output-path`. W
`--exit-on-violation`	Whether to exit the test when first failing property is found (u
`--time-limit <TIME_LIMIT>`	Maximum time to run the test; reaching the limit is treated as
`--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
`--instrument-javascript <INSTRUMENT_JAVASCRIPT>`	What types of JavaScript to instrument for coverage tracking
`--chrome-grant-permissions <CHROME_GRANT_PERMISSIONS>`	Comma-separated list of Chrome permissions to grant. Exam
`--header <KEY=VALUE>`	Extra HTTP header to send with all browser requests, in `KEY`
`--reproduce <TRACE_FILE>`	Reproduce a previous test run from a trace file, instead of ra
`--headless`	Whether the browser should run in a visible window or not
`--no-sandbox`	Disable Chromium sandboxing
`-h, --help`	Print help

bombadil browser 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.)
`--output-path-overwrite`	Overwrite any existing `trace.jsonl` at `--output-path`. W

Option	Description
`--exit-on-violation`	Whether to exit the test when first failing property is found (u
`--time-limit <TIME_LIMIT>`	Maximum time to run the test; reaching the limit is treated as
`--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
`--instrument-javascript <INSTRUMENT_JAVASCRIPT>`	What types of JavaScript to instrument for coverage tracking
`--chrome-grant-permissions <CHROME_GRANT_PERMISSIONS>`	Comma-separated list of Chrome permissions to grant. Exam
`--header <KEY=VALUE>`	Extra HTTP header to send with all browser requests, in KEY
`--reproduce <TRACE_FILE>`	Reproduce a previous test run from a trace file, instead of ran
`--remote-debugger <REMOTE_DEBUGGER>`	Address to the remote debugger's server, e.g. http://localhos
`--create-target`	Whether Bombadil should create a new tab and navigate to t
`-h, --help`	Print help

bombadil browser inspect

`bombadil inspect [OPTIONS] <TRACE_PATH>`

Argument	Description
`<TRACE_PATH>`	Path to trace.jsonl file or directory containing it

Option	Description	Default
`--port <PORT>`	Port to bind the inspect server to	1073
`--no-open`	Skip auto-opening browser	
`-h, --help`	Print help	