Quickstrom Documentation

Release 0.5.0

Oskar Wickström

Nov 10, 2022

CONTENTS

Quickstrom is an autonomous testing tool for the web. It can find problems in any type of web application that renders to the DOM. Quickstrom automatically explores your application and finds problems according to a specification. Focus your effort on understanding and specifying your system, and Quickstrom can test it for you.

Interested? Let's get started!

CHAPTER

ONE

DOCUMENTATION

If you're new to Quickstrom, start here:

- *Installation*: how to get Quickstrom running on your computer
- *Writing Your First Specification*: the entry-level tutorial

The documentation is split up into sections depending on the type of document:

- *Topics*: high-level explanations of concepts and how they fit together
- *Tutorials*: step-by-step guides focused on learning
- *How-To Guides*: short guides to achieve specific goals

CHAPTER

TWO

STAYING UPDATED

Sign up for the the newsletter.

2.1 Installation

Follow these steps to install Quickstrom locally. These are the currently supported installation methods:

2.1.1 Installing with Nix

Follow these steps to install Quickstrom using Nix.

Prerequisites

- Nix (see nix.dev for installation instructions and guides)

Installing with Nix

To install the `quickstrom` executable, use Cachix and Nix to get the executable:

```
$ cachix use quickstrom
$ nix-env -iA quickstrom -f https://github.com/quickstrom/quickstrom/tarball/main
```

If you're on Darwin, you're probably going to have problems with Firefox and Chrome from nixpkgs. You can exclude browsers from the Quickstrom environment and provide them on your own, e.g. through Homebrew. If you want to do this, override it with this command:

```
$ nix-env -iA quickstrom -f https://github.com/quickstrom/quickstrom/tarball/main \
    --arg includeBrowsers false
```

Verify that Quickstrom is now available in your environment:

```
$ quickstrom --help
```

You're now ready to *check webapps using Quickstrom*.

2.1.2 Installing with Docker

QuickStrom provides a Docker image as an easy installation method. Download the image using Docker:

```
$ docker pull quickstrom/quickstrom:0.5.0
```

Verify that Quickstrom can now be run using Docker:

```
$ docker run quickstrom/quickstrom:0.5.0 \
    quickstrom --help
```

You can now run Quickstrom checks:

```
$ docker run \
  -v $PWD/specs:/specs \
  quickstrom/quickstrom:0.5.0 \
  quickstrom -I/specs \
  check example \
  https://example.com
```

There's a lot of things going in the above command. Let's look at what each line does:

1. Uses *docker run* to execute a program inside the container
2. Mounts a host directory containing specification(s) to `/specs` in the container filesystem
3. Uses the image `quickstrom/quickstrom` with the `0.5.0` tag
4. Runs `quickstrom` with the mounted `/specs` directory as an include path
5. Checks the `example` specificiation module (i.e. `/specs/example.strom`)
6. Passes an origin URI (this could also be a file path into the mounted directory)

2.1.3 Accessing a Server on the Host

If you wish to run Quickstrom in Docker and test a website being hosted by the Docker host system you can set the URL to `localhost` (or `host.docker.internal` for MacOS).

```
$ docker run \
  --network=host \
  -v $PWD/specs:/specs \
  quickstrom/quickstrom:latest \
  quickstrom -I/specs \
  check example \
  http://localhost:3000 # or http://host.docker.internal:3000 for MacOS
```

You may have to disable HOST checking if you get "Invalid Host header" messages.

2.2 Topics

This section explains how the different parts of Quickstrom work and fit together at a high level.

2.2.1 How It Works

In Quickstrom, a tester writes *specifications* for web applications. When *checking* a specification, the following happens:

1. Quickstrom navigates to the *origin page*, and awaits the initial event (e.g. that the page has loaded).
2. Based on the current DOM state, it generates a random action to simulate user interaction. Many types of actions can be generated, e.g. clicks, key presses, focus changes, reloads, navigations.

   Only actions with their *preconditions* met can be chosen. For instance, you cannot click buttons that are currently invisible.
3. After an action has been taken, Quickstrom queries and records the state of relevant DOM elements.

   If the action takes specifies a *timeout*, Quickstrom waits the given duration for any asynchronous events to occur, e.g. that a DOM node changed.

   Steps 2 and 3 are repeated as long as the specification requires more states and actions. The recorded sequence of states and actions is called a *trace*.
4. The specification defines one or more proposition to be checked. A proposition is a logical formula, which is used to determine if the trace is *successful* or *failed*. As soon as a proposition has a definitive answer, the check stops.

Now, how do you write specifications and propositions? Let's have a look at *The Specification Language*.

2.2.2 The Specification Language

In Quickstrom, the intended behavior of a web application is described in a specification language called *Specstrom*. It's a propositional temporal logic with a functional expression language. Syntax-wise we try to keep the language close to JavaScript, although semantically it's quite different.

This document is a high-level and informal walkthrough of the language. Have a look at the paper Quickstrom: Property-based Acceptance Testing with LTL Specifications if you're interested in a more detailed description of the underlying temporal logic, called QuickLTL.

Formulae

The top level construct in Quickstrom are linear temporal logic formulae, where you can describe how the state of the web application should be now and in the future.

For example, if we want `x` to be true in the current state, and `y` in the next state, we can express it like this:

```
x && next y
```

If we want `x` to be true zero or more states until `y` becomes true, we say:

```
x until y
```

These examples are formulae, where `x` and `y` are free variables, and `next` and `until` are temporal operators.

Expressions

At a lower level we have an expression language, which is a functional language. We can do basic stuff like arithmetic and comparisons:

```
x + y

x >= (y - z)
```

Boolean expressions are automatically lifted to the formulae level. Here's an expression (`x > y`) which is lifted into a formula (`next ...`):

```
next (x > y)
```

Selectors

A selector is a CSS selector written inside backticks, which evaluates to a list of elements. For example, the following selector evaluates to a list of all button elements in a given state:

```
`button`
```

We can iterate over the list of elements and refer to attributes, properties, CSS styles, and more:

```
for b in `button` { b.textContent }
```

The example above evaluates to a list of strings.

If we're only interested in the first element, we can directly select from it, as long as the list is not empty:

```
`button`.textContent
```

A selector's evaluated is state-dependent, meaning that it can evaluate to different lists of elements depending on in which state it is evaluated.

In the following example, we might not get the same list for both selectors, so it could be true:

```
length(`button`) == 1 && next (length(`button`) == 2)
```

Elements

The elements we get by evaluating selectors are objects. We can refer to various things in those objects to read relevant state from the DOM:

enabled
: is the element enabled?

visible
: is the element visible?

interactable
: is the element interactable (e.g. clickable)?

active
: is the element active?

`classList`
: a list of strings, based on the `class` attribute

`css`
: a nested object with computed styles

`attributes`
: a nested object with HTML element attributes

As a fallback, any other key is evaluated as a property on the corresponding runtime object of the element.

The Quickstrom expression `` `button`.textContent `` corresponds to the following JavaScript expression:

```
document.querySelector("button").textContent
```

Let Bindings

In expressions and in formulae, we bind values to names using `let`. The general form is:

```
let name = expression; body
```

If we need many bindings, we can put them on separate lines:

```
let foo = 1;
let bar = 2;
let baz = 3;
...
```

Let is also supported as a top-level construct in source files.

Lazy Bindings

When expressions in let bindings are state-dependent, like those involving selectors, we don't want the expression to be evaluated when bound. Instead we annotate the binding using a tilde prefix, meaning it's a lazy binding:

```
let ~myButtons = `.btn`;
```

The expression, in this case `` `.btn` ``, is evaluated when another expression refers to `myButtons` and is itself evaluated. Different evaluations of `myButtons` may result in different values, depending in which state the evaluation occurs. For example, the formula `next myButtons` might not be equivalent to `next (next myButtons)`.

Propositions

When testing web apps using Quickstrom, we define *propositions* and ask Quickstrom to check them for us. A proposition is a formula defined at the top level. Useful propositions are state-dependent, so they are always bound lazily in practice.

Let's say we have some proposition bound to `prop`. We could check it like so:

```
import quickstrom;

let ~prop = ...;

check prop with * when loaded?;
```

Read more about the `check` statement in the *Check* section.

Temporal Operators

In Quickstrom specifications, there are a bunch of built-in temporal operators:

- `next`
- `always`
- `until`

There are also utility operators defined using the built-in temporal operators:

`unchanged`

Let's go through the operators and utility functions provided by Quickstrom with some more examples!

Next

The formula `next p` says that the formula `p` is true in the next state.

But which state is "the next state"? It depends on which is the current state. Temporal operators are always in relation to the current state.

Always

The formula `always p` says that the formula `p` is true in the current and all subsequent states.

As an example, in the following proposition we check that the heading is always `"Home"`:

```
let ~title = `h1`.textContent;
let ~prop = always (title == "Home");
```

Until

The formula `p until q` says that the formula `p` is true at least until the formula `q` is true.

Note:

- It doesn't matter if `p` is true or false once `q` is true. If we wanted that kind of exclusiveness, we could say `p until (q && not p)`.
- `q` can be true in the current state, in which case `p` never has to be true.
- `q` only has to be true in one state, it doesn't have to stay true forever. If we want it true forever, we could say `p until (always q)`.

In the following example, we check that a loading indicator is shown until the page title is set correctly:

```
let ~title = `h1`.textContent;
let ~loading = `.loading`.textContent;
let ~prop = (loading == "Loading...") until (title == "Home");
```

Unchanged

The formula `unchanged p` says that `p` in the current state is equal to `p` in the next state. Or in other words, that `p` doesn't change from this state to the next.

This operator is useful when expressing state transitions, specifying that a certain queried value should be the same both before and after a particular transition.

For instance, let's say we have a bunch of top-level definitions, all based on DOM queries, describing a user profile:

```
let ~userName = ...;

let ~userProfileUrl = ...;
```

We can say that the user profile information should not change in a transition `t` by passing an array of those values:

```
let ~t = unchanged [userName, userProfileUrl]
    && ... // actual changes in transition
    ;
```

Actions

We must instruct Quickstrom what actions are allowed. Actions are declared at the top level using the `action` keyword.

```
action launchTheMissiles! = click!(`#launch`);
```

By convention, actions are suffixed with an exclamation mark. Events on the other hand are suffixed with a question mark, but still declared using the `action` keyword:

```
action launched? = changed?(`#launch-status`)
  when `#launch-status`.textContent == "Launched!";
```

Built-in Actions

The following actions and events are provided in the Quickstrom library:

- `click!`
- `doubleClick!`
- `clear!`
- `focus!`
- `keyPress!`
- `enterText!`
- `enterTextInto!`
- `noop!`
- `changed?`
- `loaded?`

Note: Support for more actions should be added.

Action Preconditions

Actions can be constraint to only be applicable under certain preconditions. We use the *when* construct to express a precondition:

```
action launchTheMissiles! = click!(`#launch`) when canLaunch;
```

Many of the built-in actions in Quickstrom already have useful preconditions set, like *click!* only be applicable on elements that are interactable and enabled. This means that we don't have to specify such basic preconditions. It's more likely that preconditions will be domain-specific rules, if required at all.

Event Postconditions

Similar to action preconditions are event postconditions. They are used to declare an event that is only valid under certain conditions.

For instance, a DOM element might have changed, but only if it's changed in a certain way we considered it a specific event. The *launched?* event we saw earlier is defined using a postcondition:

```
action launched? = changed?(`#launch-status`)
  when `#launch-status`.textContent == "Launched!";
```

Check

We need to tell Quickstrom how to check a web application against a specification. We do that using the `check` statement, which has this general form:

```
check <props> with <actions> when <initial event>;
```

The placeholders work this way:

<props>
: This is wildcard matcher on all bindings in the current file. You can either literally refer to the propositions you want (e.g. `prop`), or use a star to match against multiple propositions (e.g. `prop_*`).

<actions>
: Also a wildcard matcher, matching on actions declared in the current file. In many cases this is just `*`.

<initial event>
: The name of the initial event. The checker waits until this event occurs before it starts performing actions. In many web applications scenarios it will be `loaded?`, but it might also be something more specialized.

As an example, we might end a specification with the following statement:

```
check prop* with * when loaded?;
```

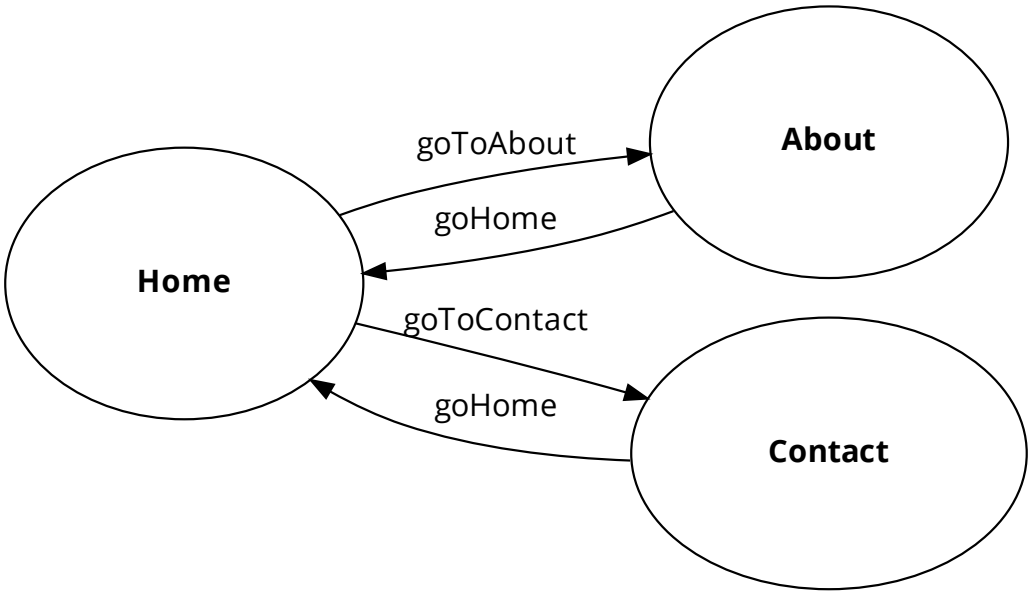
State Machine Propositions

A powerful way of writing specifications is by expressing them as state machines. A transition is expressed as an assertion about the current state and another assertion about the next state. The state machine proposition says that one of the transitions are always taken.

Here's an example based on a simple website with three pages:

```
let ~title = `h1`.textContent;

// Transitions

let ~goToAbout = title == "Home" && next title == "About";

let ~goToContact = title == "Home" && next title == "Contact";

let ~goHome = title != "Home" && next title == "Home";

// Proposition

let ~prop = always (goToAbout || goToContact || goHome);
```

The `goToAbout`, `goToContact`, and `goHome` transitions specify how the title of the page changes, and the `prop` thus describes the system as a state machine. It can be visualized as follows:



Source Files

Specifications are written in source files with a `.strom` file extension. A file is a *module*, and the module name is the filename without the `.strom.` extension.

Other modules can be imported using the `import <module name>;` syntax at the top of a module. For instance, if we have a file `foo.strom` with the following contents:

Listing 1: foo.strom

```
let x = 1;
```

We can import the `foo` module from the `bar` module and refer to its bindings and actions:

Listing 2: bar.strom

```
import foo;

let y = x + 1;
```

The module system is very rudimentary. It works similarly as C header files with include guards. Bindings from transitively imported modules are also available. Continuing on the example above, if a third module imported the `bar` module, the `x` binding would also be in scope in `baz`.

Include Paths

Quickstrom has a list of *include paths*, i.e. directories in which it tries to find the files corresponding to imported modules. The current working directory is implicitly an include path.

2.2.3 Checking

To check a web application against a specification, use the `quickstrom check` command. Supply the module name of the specification along with the origin URL.

```
$ quickstrom check \
   spec-module-name \
   http://example.com
```

The origin can also be a local file:

```
$ quickstrom check \
   spec-module-name \
   /path/to/my/webapp.html
```

Cross-Browser Testing

Quickstrom currently supports these browsers:

- Firefox (`firefox`)
- Chrome/Chromium (`chrome`)

Unless specified, the default browser used is Firefox. To override, use the `--browser` option and set the appropriate browser when running the `check` command:

```
$ quickstrom check \
   --browser=chrome \
   ... # more options
```

You can also override which binary it uses when launching the browser:

```
$ quickstrom check \
   --browser=chrome \
   --browser-binary=/usr/bin/google-chrome-stable \
   ... # more options
```

2.2.4 Reporters

After a Quickstrom check completes, one or more *reporters* run. They report the result of the check in different formats. The following reporters are available:

- `console`
- `html`
- `json`

Invoke reporters by passing the `--reporter=<NAME>` option to the `check` command.

Console

The console reporter is invoked by default. It prints a trace and summary to the console when a check fails. The trace contains information about the state of queried elements at each state in the behavior, along with the actions taken by Quickstrom.

HTML

The HTML reporter creates a report for web browsers in a given directory. The report is an interactive troubleshooting tool based on state transitions, showing screenshots and overlayed state information for the queried elements.

To set the directory to generate the report in, use the option `--html-report-directory=<DIR>`.

Tip: If you run Quickstrom using Docker, make sure to generate the HTML report in a mounted directory so that you can access it from the host.

JSON

The JSON report works similarly to the HTML report, except it generates only a JSON file and screenshots, no HTML files. In the report directory you'll find a file `report.json` that you can work with.

To set the directory to generate the report in, use the option `--json-report-directory=<DIR>`.

2.2.5 Troubleshooting

This documents collects some common problems and tips on how to identify what's not working correctly.

If you're troubleshooting a failing Quickstrom check, make sure to enable debug logs:

```
$ quickstrom --log-level=DEBUG check ...
```

Also, the underlying Specstrom interpreter outputs its log file in the current working directory, called `interpreter.log`. Its location can be changed using the `--interpreter-log-file=...` option.

Finally, the Webdriver log can be stored by specifying `--driver-log-file=...`.

2.3 Tutorials

2.3.1 Writing Your First Specification

In this tutorial we'll specify and check an *audio player* web application using Quickstrom.

The tutorial assumes you're running on a Unix-like operating system and that you have Docker installed. You may run this using *other installation methods*, but all commands in this document are using Docker.

Open up a terminal and create a new directory to work in:

```
$ mkdir my-first-spec
$ cd my-first-spec
```

Installing with Docker

In this tutorial you need a working installation of Docker. Head over to docker.com and set it up if you haven't already.

Next, pull the QuickStrom image using Docker:

```
$ docker pull quickstrom/quickstrom:0.5.0
```

Downloading the Audio Player

The web application we're going to test is already written. Download it using `curl`:

```
$ curl -L https://github.com/quickstrom/quickstrom/raw/main/docs/source/_static/
↪audioplayer/audioplayer.html -o audioplayer.html
```

If you don't have `curl` installed, you can download it from this URL using your web browser. Make sure you've saved it our working directory as `audioplayer.html`.

```
$ ls
audioplayer.html
```

OK! We're now ready to write our specification.

A Minimal Specification

We'll begin by writing a specification that always makes the check pass. Create a new file `audioplayer.strom` and open it in your text editor of choice:

```
$ touch audioplayer.strom
$ $EDITOR audioplayer.strom
```

Type in the following in the file and save it:

```
1 import quickstrom;
2
3 action ~playOrPause! = click!(`.play-pause`);
4
5 let ~proposition = true;
6
7 check proposition with * when loaded?;
```

A bunch of things are going on in this specification. Let's break it down line by line:

- **Line 1:** We import the Quickstrom module. This is where we find definitions for DOM queries, actions, and logic. We also import *Maybe* which we'll need later on.
- **Line 3:** Our actions specify what Quickstrom should try to do. In this case, we want it to click the play/pause button.
- **Line 5:** In the `proposition`, we specify what it means for the system under test to be valid. For now, we'll set it to `true`, meaning that we require only a single state, and that *any* such state is considered valid.
- **Line 7:** The `check` statement tells Quickstrom how to test our application. We ask it to check our defined `proposition`, with all declared actions, once the `loaded?` event has occured.

Running a Test

Let's run some tests! Launch Quickstrom from within your `my-first-spec` directory:

```
$ docker run --shm-size=1g --rm \
  -v $PWD:/my-first-spec \
  quickstrom/quickstrom:0.5.0 \
  quickstrom -I/my-first-spec check \
  audioplayer \
  /my-first-spec/audioplayer.html \
  --browser=chrome
```

You should see output like the following:

```
The test passed.
```

Cool, we have it running! So far, though, we haven't done much testing. Quickstrom doesn't do more than the specification requires, and right now any initial state is good enough, so it doesn't perform any actions. Let's make our specification say something about the audio player's intended behavior.

Refining the Proposition

Our system under test (`audioplayer.html`) is very simple. There's a button for playing or pausing the audio player, and there's a time display.

Our specification will describe how the player should work. Informally, we state the requirements as follows:

- Initially, the player should be `paused`
- When `paused`, and when the play/pause button is clicked, it should transition to the `playing` state
- When in the `playing` state, the time display should reflect the progress with a ticking minutes and seconds display
- When `playing`, and when the play/pause button is clicked, it should go to the `paused` state, and time should not change
- In the `paused` state, the button should say "Play"
- In the `playing` state, the button should say "Pause"

Let's translate those requirements to a formal specification in Quickstrom.

Begin by defining two element helpers, extracting the text content of the play/pause button, and extracing and parsing the time display text. The time is represented as total number of seconds in our specification, making it easier to compare.

Place these just after the imports section in `audioplayer.strom`:

```
let ~buttonText = `.play-pause`.textContent;

let ~timeInSeconds =
  let [minutes, seconds] = split(":", `.time-display`.textContent);
  parseInt(minutes) * 60 + parseInt(seconds);
```

Next, we'll define the two states as booleans:

```
let ~playing = buttonText == "Pause";

let ~paused = buttonText == "Play";
```

We also need to declare the actions a bit more precisely. Change to existing action declartion to the following:

```
action ~pause! = click!(`.play-pause`) when playing;

action ~play! = click!(`.play-pause`) when paused;
```

Finally, we'll change the `proposition`. Remove `true` and type in the following code:

```
let ~proposition =
  let ~play = ...;
  let ~pause = ...;
  let ~tick = ...;
  paused && (always {20} (play || pause || tick));
```

Note: The `...` parts aren't valid ecpressions, but we'll replace them with valid ones in the next section.

The last line in our proposition can be read in English as:

> Initially, the record player is paused. From that point, one can either play or pause, or the time can tick while playing, all indefinitely.

OK, onto adding the missing parts!

The Missing State Transitions

We have a bunch of `...` placeholders in our state transition formulae. Let's fill them in!

The definition `play` describes a transition between `paused` and `playing`:

```
let ~play =
  paused
    && nextT playing
    && unchanged(timeInSeconds);
```

OK, so what's going on here? We specify that the current state is `paused`, and that the next state is `playing`. That's how we encode state transitions. We also say that the time shouldn't change.

Note: We need to use `nextT` instead of `next` here, because we don't want to force another state being read. If there is a next state available, we say that it should be `playing`, otherwise we default to true. That's what the `T` in `nextT` means.

The `pause` transition should look similar:

```
let ~pause =
  playing
    && nextT paused
    && unchanged(timeInSeconds);
```

Finally, we have the `tick`. When we're in the `playing` state, the time changes on a `tick`. The time should be monotonically increasing, so we compare the current and the next time:

```
let ~tick =
  playing
    && nextT playing
    && (let old = timeInSeconds; nextT (old < timeInSeconds));
```

That's it! Your proposition should now look something like this:

```
let ~proposition =
  let ~play =
    paused
      && nextT playing
      && unchanged(timeInSeconds);

  let ~pause =
    playing
```

(continues on next page)

(continued from previous page)

```
      && nextT paused
      && unchanged(timeInSeconds);

  let ~tick =
    playing
      && nextT playing
      && (let old = timeInSeconds; nextT (old < timeInSeconds));

  paused && (always {20} (play || pause || tick));
```

Let's run some more tests.

Catching a Bug

Run Quickstrom again, now that we've fleshed out the specification:

```
$ docker run --shm-size=1g --rm \
  -v $PWD:/my-first-spec \
  quickstrom/quickstrom:0.5.0 \
  quickstrom -I/my-first-spec check \
  audioplayer \
  /my-first-spec/audioplayer.html \
  --browser=chrome
```

You'll see a bunch of output, involving shrinking tests and more. It should end with something like the following:

```
Transition #1

Actions and events:

- click('33a4c299-a382-44c2-870e-b48335f9c23a')

State difference:

`.play-pause`

| 33a4c299-a382-44c2-870e-b48335f9c23a | 33a4c299-a382-44c2-870e-b48335f9c23a |
| enabled: true                        | enabled: true                        |
| interactable: true                   | interactable: true                   |
| textContent: "Play"                  | textContent: "Play"                  |
| visible: true                        | visible: true                        |


`.time-display`

| bd797365-5239-4a9b-9976-942288bd227a | bd797365-5239-4a9b-9976-942288bd227a |
| textContent: "00:00"                 | textContent: "00:00"                 |
```

Look, we've found our first bug using Quickstrom! It seems clicking the play/pause doesn't do anything. It should change the label to "Pause" to indicate it's in the playing state.

The problem is the button text. Open up `audioplayer.html`, and change the following function called `playPauseLabel`:

```
function playPauseLabel(state) {
    let label;
    switch (state) {
    case "playing":
        label = "Pause";
    case "paused":
        label = "Play";
    }
    return label;
}
```

It should be:

```
function playPauseLabel(state) {
    switch (state) {
    case "playing":
        return "Pause";
    case "paused":
        return "Play";
    }
}
```

Are we done? Is the audio player correct? Not quite.

Transitions Based on Time

The audio player transitions between states mainly as a result of user action, but not only. A `tick` transition (going from `playing` to `playing` with an increased time) is triggered by *time*.

In addition to our `play!` and `pause!` actions, we'll add a `wait!` action. It does nothing for at most two seconds, until either something happens (a `tick?` event) or a timeout:

```
action ~wait! = noop! timeout 2000;
```

But this means we have to allow nothing to change. This is often called a *stutter state*. Let's do that, but only in combination with a `noop!` action. Add one more state transition to the proposition:

```
let ~proposition =
  ...
  let ~wait =
    nextT (contains(noop!, happened))
      ==> unchanged([timeInSeconds, playing]);

  paused && (always {20} (
    play
    || pause
    || tick
    || wait
  ));
```

Run another check by executing the same command as before:

```
$ docker run --shm-size=1g --rm \
  -v $PWD:/my-first-spec \
  quickstrom/quickstrom:0.5.0 \
  quickstrom -I/my-first-spec check \
  audioplayer \
  /my-first-spec/audioplayer.html \
  --browser=chrome
```

You should see output such as the following:

```
parseInt could not parse: "NaN"
  ParseInt @ /my-first-spec/audioplayer.strom:15:2
  <fun> @ /nix/store/xradz0aav0ijajw91x1vqfqsprgipfhx-specstrom/share/ulib/control.
↪strom:30:21
```

Whoops, look at that! It crashes because the time display shows "NaN", which is definitely not intended behavior. Open up `audioplayer.html`, and change the following lines near the end of the file:

```
case "pause":
    return await inPaused();
```

They should be:

```
case "pause":
    return await inPaused(time); // <-- this is where we must pass in time
```

Rerun the check using the same `quickstrom` command as before. It passes!

Summary

Congratulations! You've completed the tutorial, created your first specification, and found multiple bugs.

Have we found all bugs? Possibly not. This is the thing with testing. We can't know if we've found all problems. However, with Quickstrom you can increase your confidence in the correctness of your web app, especially if you continously run tests on it.

This tutorial is intentionally fast-paced and low on theory. Now that you've got your hands dirty, it's a good time to check out *The Specification Language* to learn more about the operators in Quickstrom.

2.4 How-To Guides

Achieve specific goals with these cookbook-style guides. They're not as thorough as the *Tutorials* and normally expect you to know how to set up and run Quickstrom already.

2.4.1 Broken Links

This is simple specification used for finding broken internal links. External links are not followed. It's based on Domen Kožar's gist.[1]

```
import quickstrom;

// Only links beginning with a slash and ending
// with .html are followed. We could also use
// an absolute base URL, e.g:
//
//     a[href^='https://example.com']
//
action ~navigate! =
  click!(`a[href^='/'][href$='.html']`);

// We're interested in finding links that lead
// to pages rendered with these status codes in
// the heading.
let patterns = [ "404", "500" ];

// In our system's error pages, status codes
// are rendered in an <h1> element.
let ~heading =
  let h1 = first(`h1`);
  h1.textContent when h1 != null;

// Check if the heading has any of the error
// codes in the text.
let ~hasErrorCode =
  exists p in patterns {
    contains(p, heading)
  };

// This is the safety property. At no point,
// following only internal links, should we
// see error codes.
let ~proposition = always (not hasErrorCode);

check proposition with * when loaded?;
```

Tweak the patterns and the predicate to fit your use case. You might not have status codes rendered, but texts like "Page not found" or "Access denied".

[1] Domen Kožar wrote the original specification for Cachix using one of the first versions of Quickstrom, and published it along with a GitHub Actions setup: https://gist.github.com/domenkozar/71135bf7aa6d50d6911fb74f4dcb4bad

2.4.2 Testing in GitHub Actions

Quickstrom can be run in a continuous integration (CI) workflow to find problems early. Here's a configuration for GitHub Actions that checks a website on every commit to the `main` branch. It's based on Domen Kožar's gist.[1]

```
name: "Quickstrom integration tests"
on:
  push:
    branches:
      - main
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2.3.4

    # We use `install-nix-action` and `cachix-action` to quickly install
    # Quickstrom from a binary cache.
    - uses: cachix/install-nix-action@v18
    - uses: cachix/cachix-action@v12
      with:
        name: quickstrom
    - run: nix-env -iA quickstrom -f https://github.com/quickstrom/quickstrom/tarball/0.
↪5.0

    # Now, run tests! This assumes there's a file called
    # `example.strom` in the root of the GitHub repository.
    - run: quickstrom check example https://example.com --reporter=html --html-report-
↪directory=report

    # Finally, we archive HTML report as an artifact. This
    # can be downloaded an inspected after failed checks.
    - name: Archive test results
      uses: actions/upload-artifact@v3
      with:
        name: test-report
        path: report
```

Replace the placeholder paths and URLs.

Next Steps

- You might want to run tests in Chrome instead. See *Checking* for instructions on using other browsers.
- If you'd like to check multiple specs and in multiple browsers, see matrix configurations in the GitHub Actions documentation.

[1] Domen Kožar wrote the original GitHub Action configuration for Cachix: https://gist.github.com/domenkozar/71135bf7aa6d50d6911fb74f4dcb4bad

2.5 FAQ

Here are some frequently asked questions about Quickstrom:

2.5.1 Isn't this just property-based testing for web applications?

Quickstrom definitely is a form of property-based testing (PBT), but it's not *only* that. Being specifically designed for testing web applications, Quickstrom can reduce the amount of work you need to do in order to test properties of your system:

- Quickstrom discovers and performs actions automatically
- You specify only the properties you care about, and you don't have to write a fully functional model
- Quickstrom aims to (in the future) perform fault injection automatically, such as delaying, cancelling, or manipulating XHR responses, run in concurrent tabs, manipulate cookies or web storage, etc

It might be useful to think of Quickstrom as a mix of PBT, black-box browser testing, and a specification system like TLA+. One aim is "to be the Jepsen for web applications."

2.5.2 Why should I use Quickstrom instead of a model-based property test?

You might argue that this is just property-based testing, and that you could do this with state machine testing. And you'd be right! Similar tests could be written using a state machine model, WebDriver, and property-based testing.

With Quickstrom, however, you don't have to write a model that fully specifies the behavior of your system. Instead, you describe the most important state transitions and leave the rest unspecified. You can gradually adopt Quickstrom and improve your specifications over time.

Furthermore, in problem domains where there's lots of of *essential complexity*, models tend to become as complex. For example, it's often hard to find a naive implementation for your model when your modelling a business system with a myriad of arbitrary rules.

Finally, by using linear temporal logic, we can express safety and liveness properties in specifications. This is something that you'd have to build yourself on top of regular property tests.