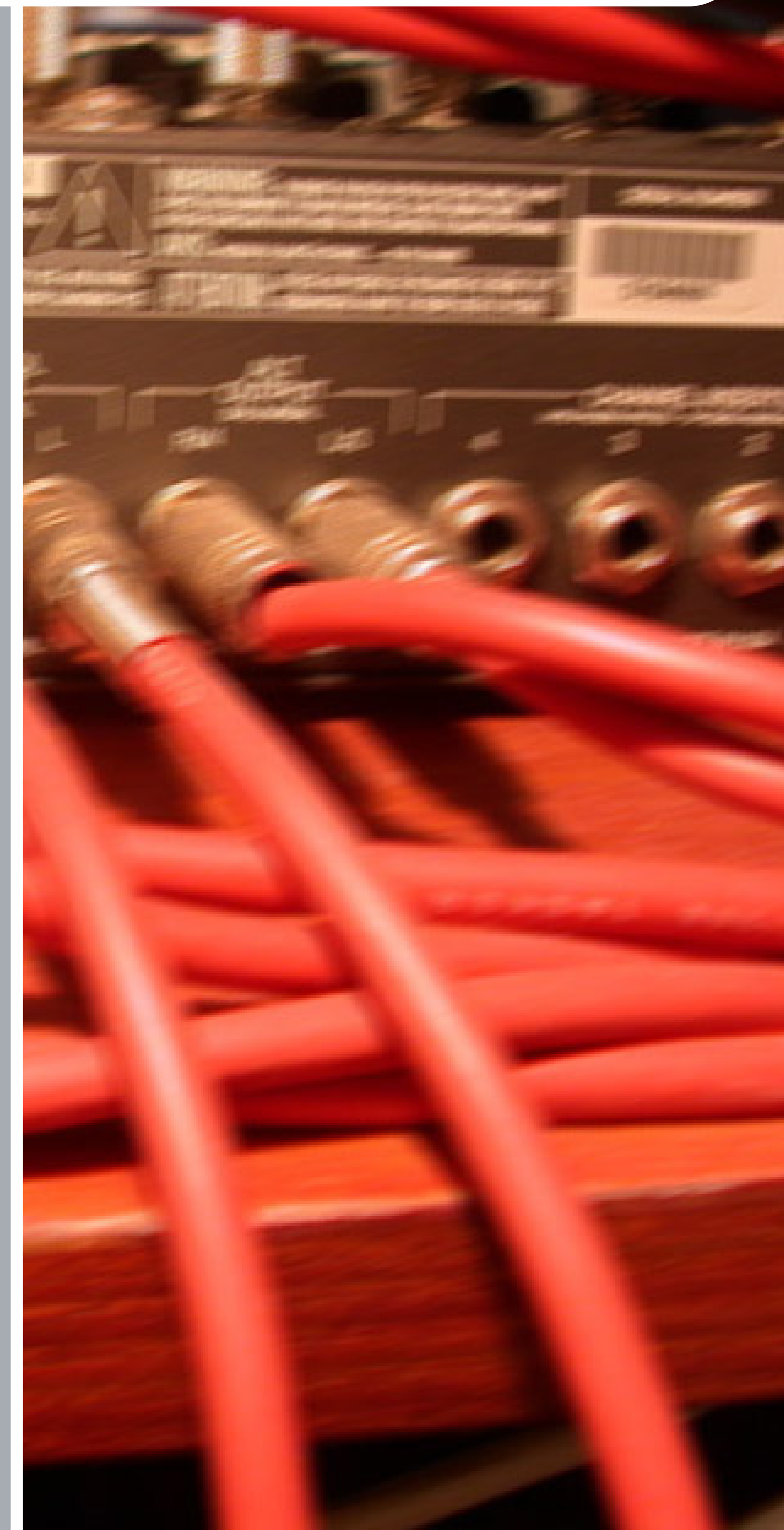
Ruby for the Web

The Arrow Web Application Framework

Michael Granger <ged@FaerieMUD.org>

O'Reilly Open Source Convention 2004

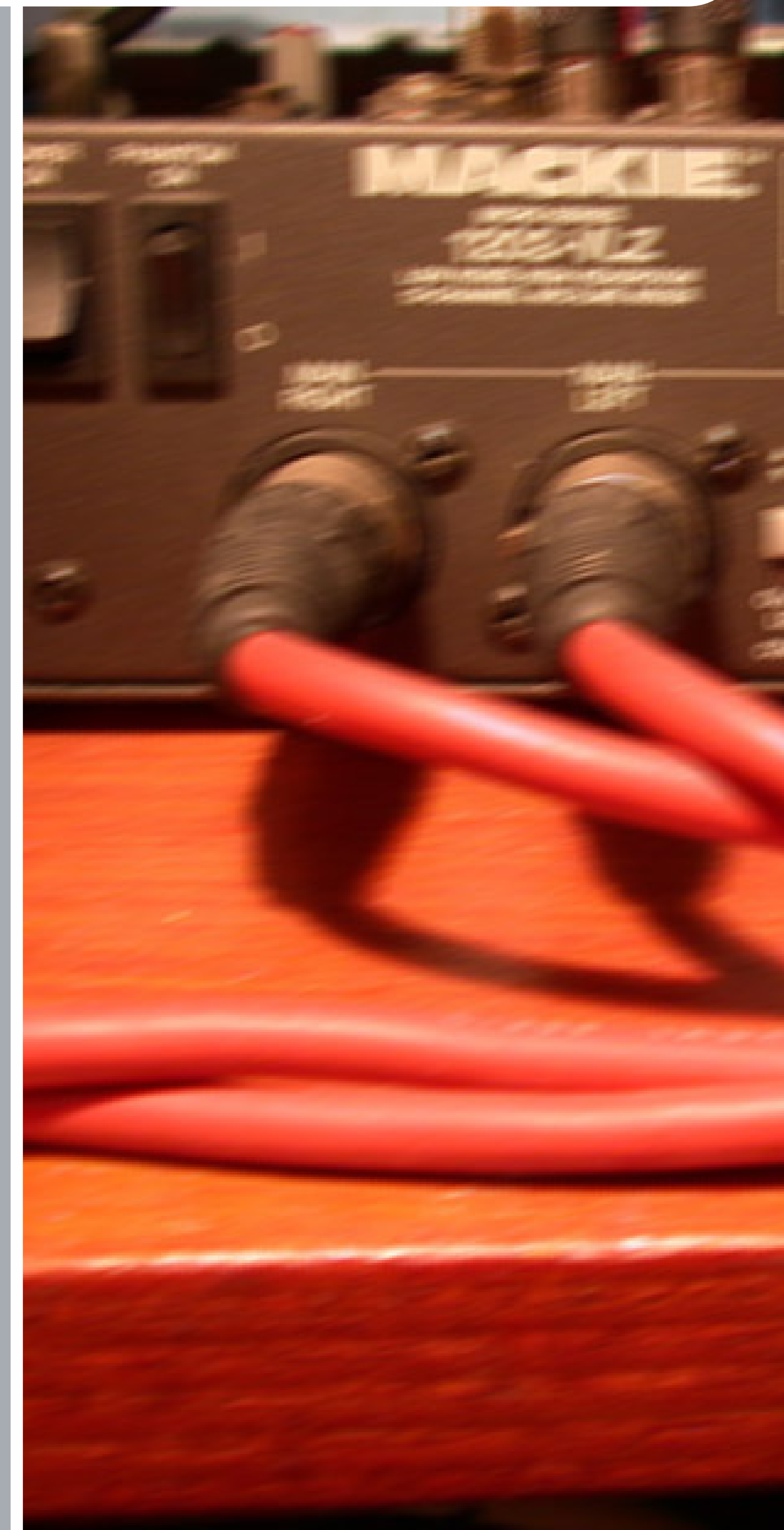
What is Arrow?



- A web application framework for building mod_ruby applications
- Designed to make development under Apache easy and more fun
- …but without sacrificing the power of the native Apache API

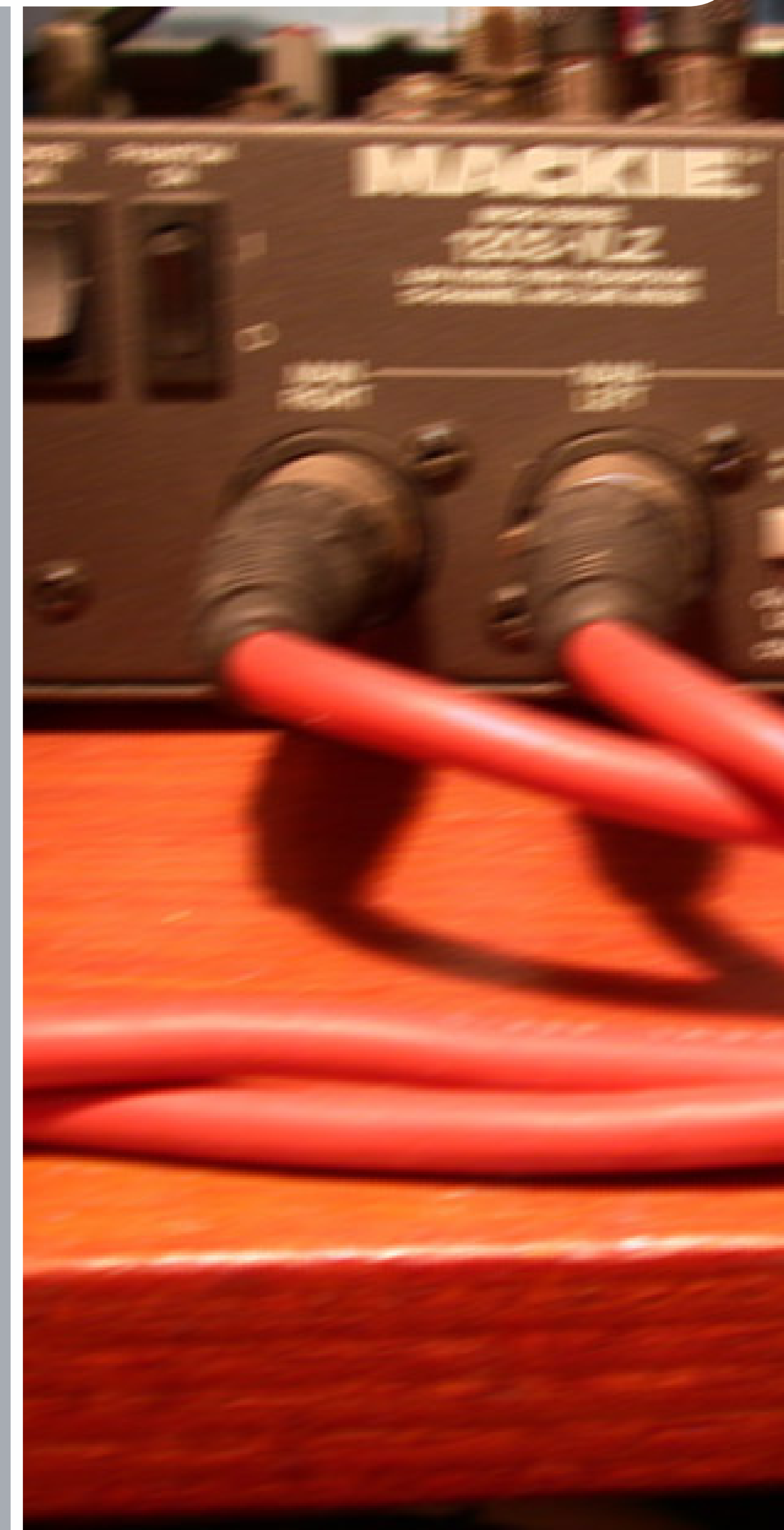
Why *Another* Framework?

- At the time, there were no frameworks for mod_ruby
- Apache-based vs. standalone server
- Feature sets lacked important functionality
- Extensibility/Customization



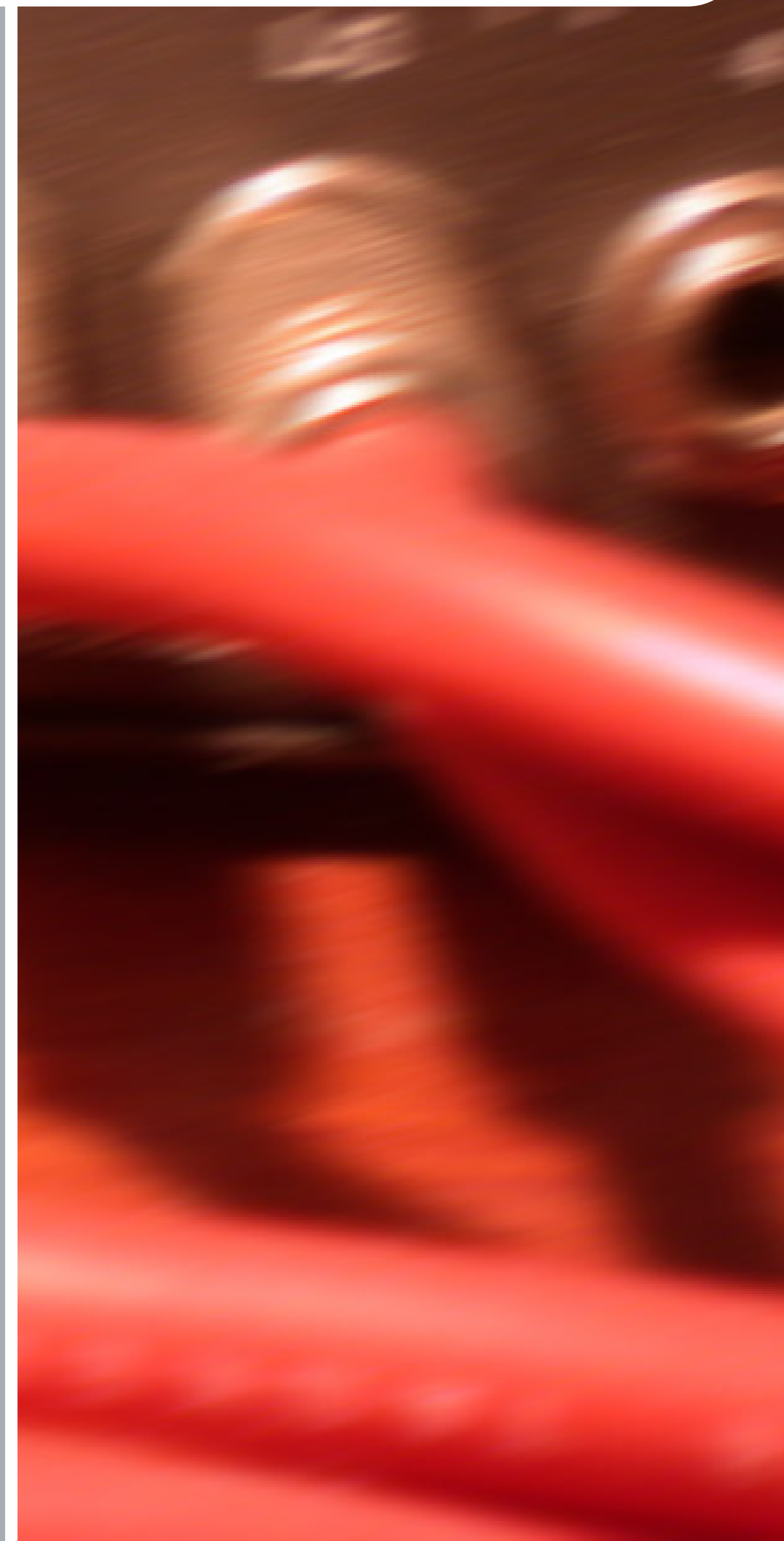
Why *Another* Framework? (cont.)

- Today, there are several frameworks that meet most of our criteria.
- …but we still like Arrow better.
- You might too, but if not, that’s okay.



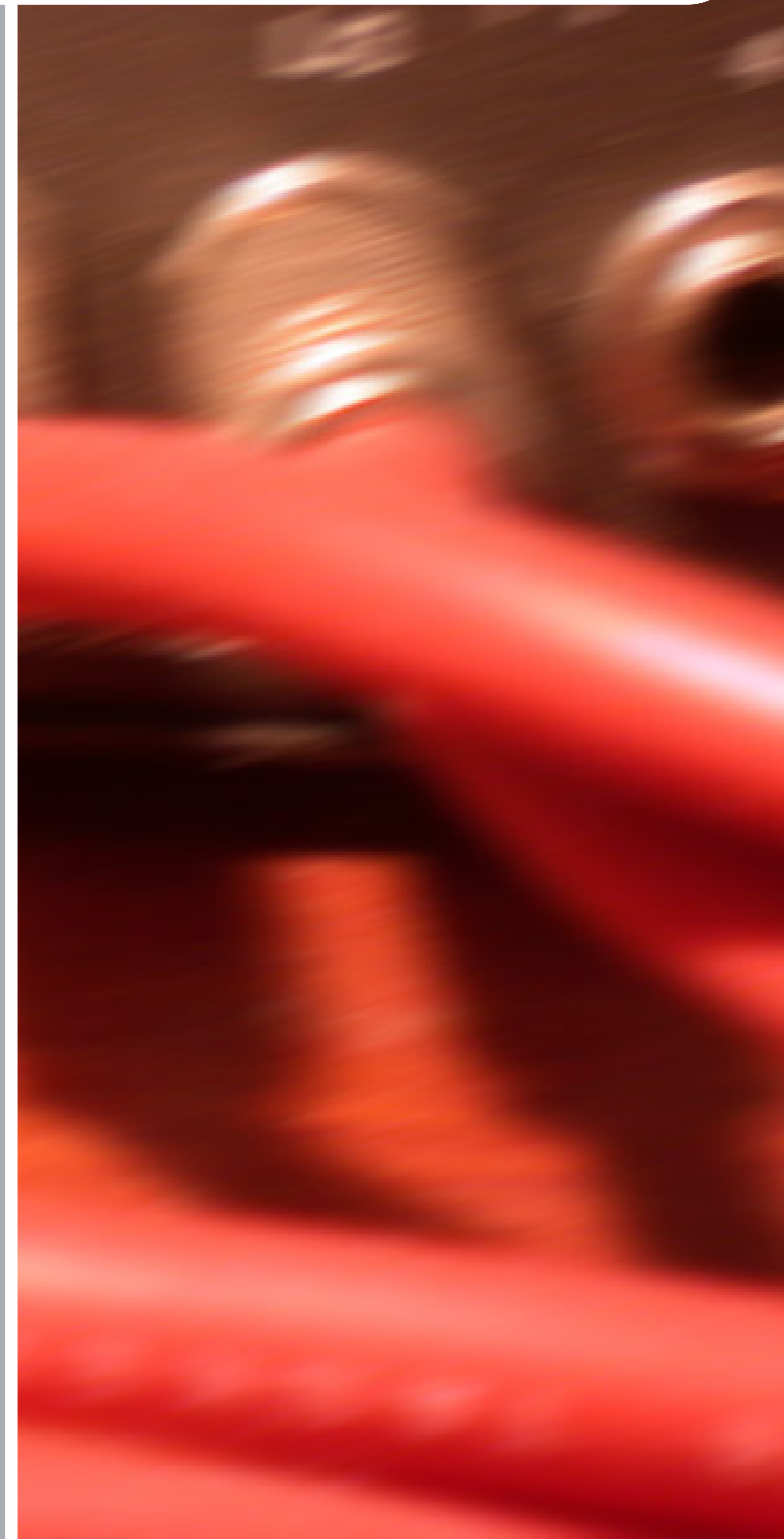
Arrow Design Goals

- Motivation: Create a framework for creating applications for clients
  - Create once, customize many times
  - Components allow mix-and-match across applications
  - Easily customizable to client's particular needs



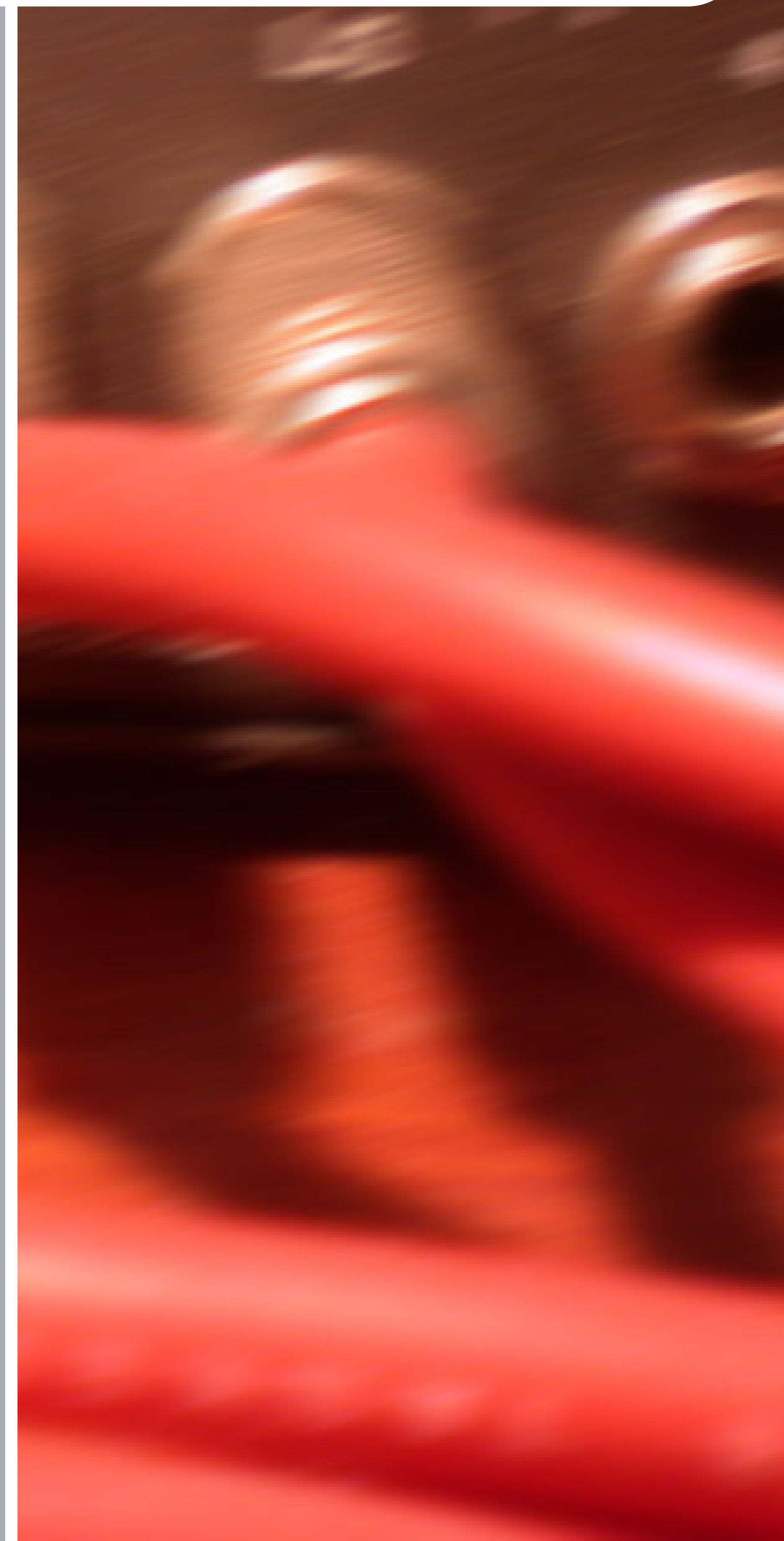
Arrow Design Goals

- Ease of development features
  - Provide access to the Apache API but as easy as CGI (or easier)
  - No server-restart cycle
  - Auto-reloadable everything



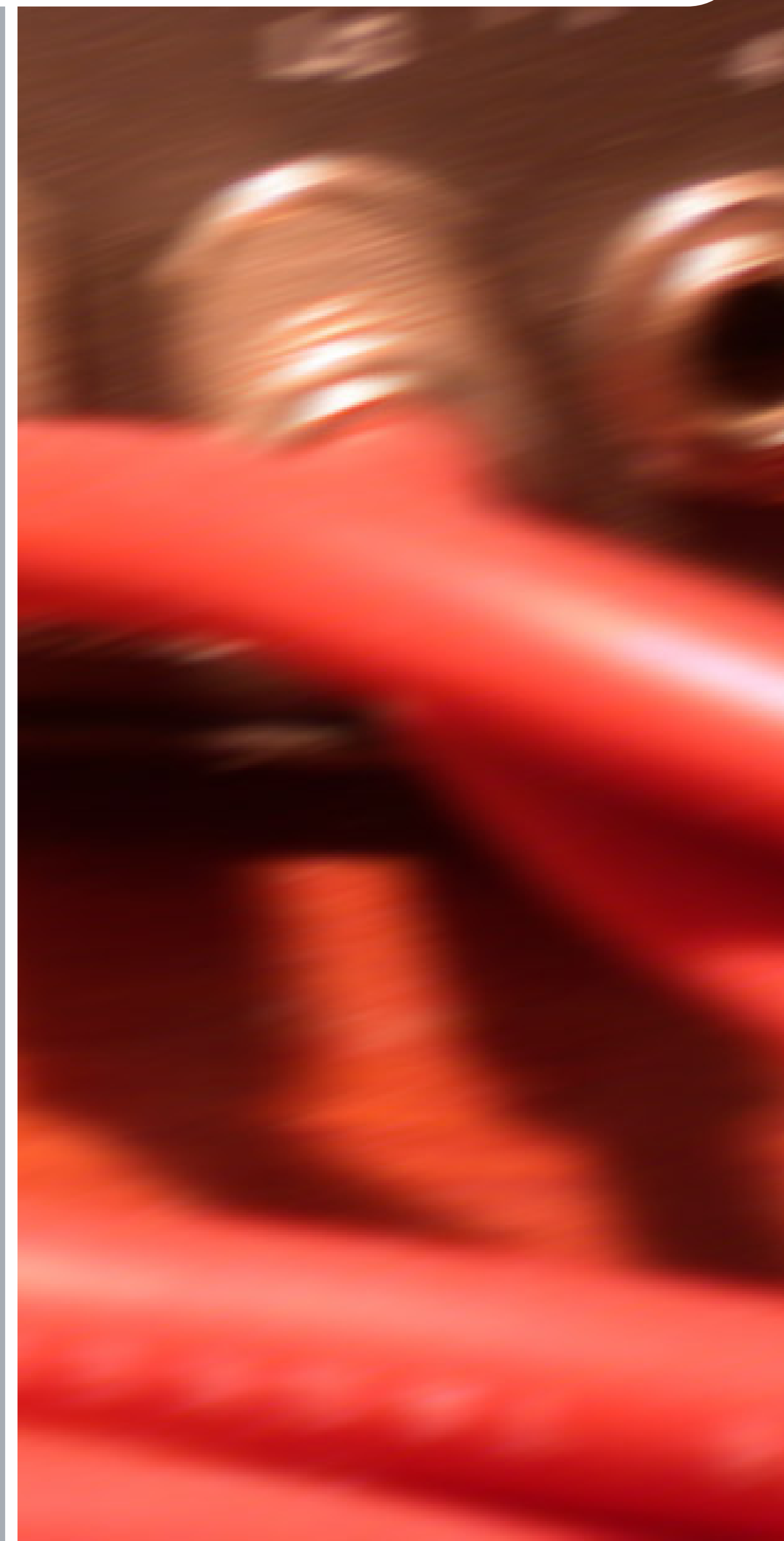
Arrow Design Goals (cont.)

- Other niceties
  - Pluggable session management
  - Argument validation/untainting
  - Graceful error-handling



Arrow Design Goals (cont.)

- MVC Architecture
  - View: Strong separation of code from HTML
  - Component-driven modular controllers
  - Model-agnostic



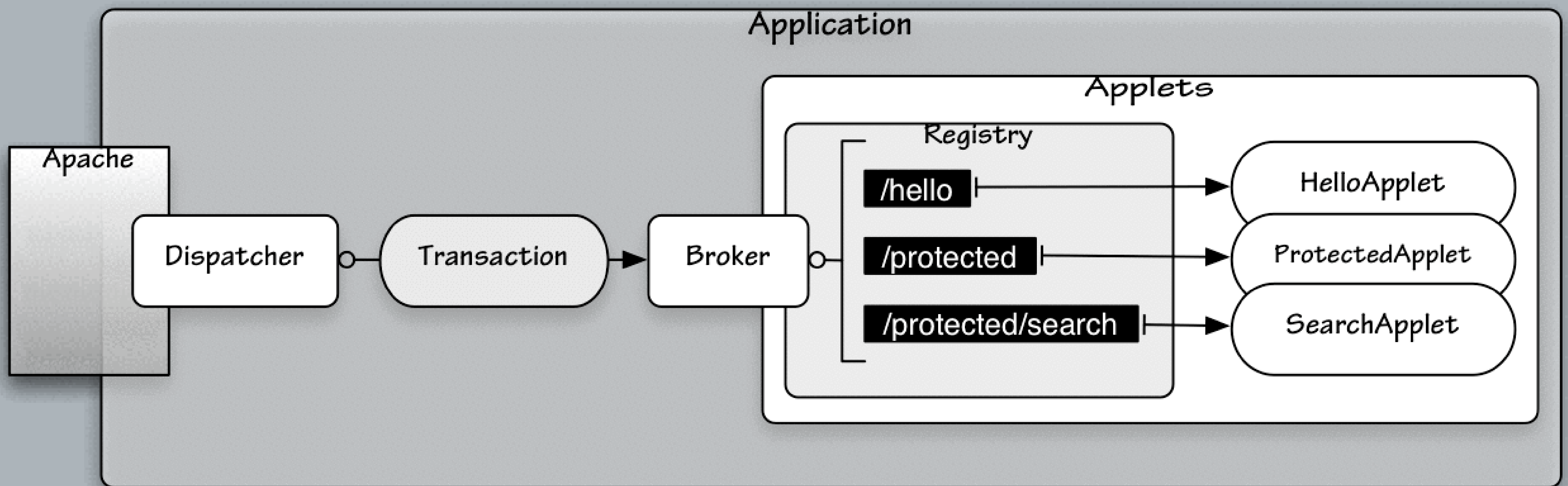
Overview

- *Applications* are made up of one or more *applets*
- Each *applet* handles a single mode or group of *actions* within the application
- *Actions* are single use-cases or verbs within an applet

Overview (cont.)

- An example:
  - **Application:** Content Management System
  - **Applets:** add, edit, search, delete
  - **Actions**
    - **add:** form, upload, confirm, save
    - **delete:** choose, confirm, remove

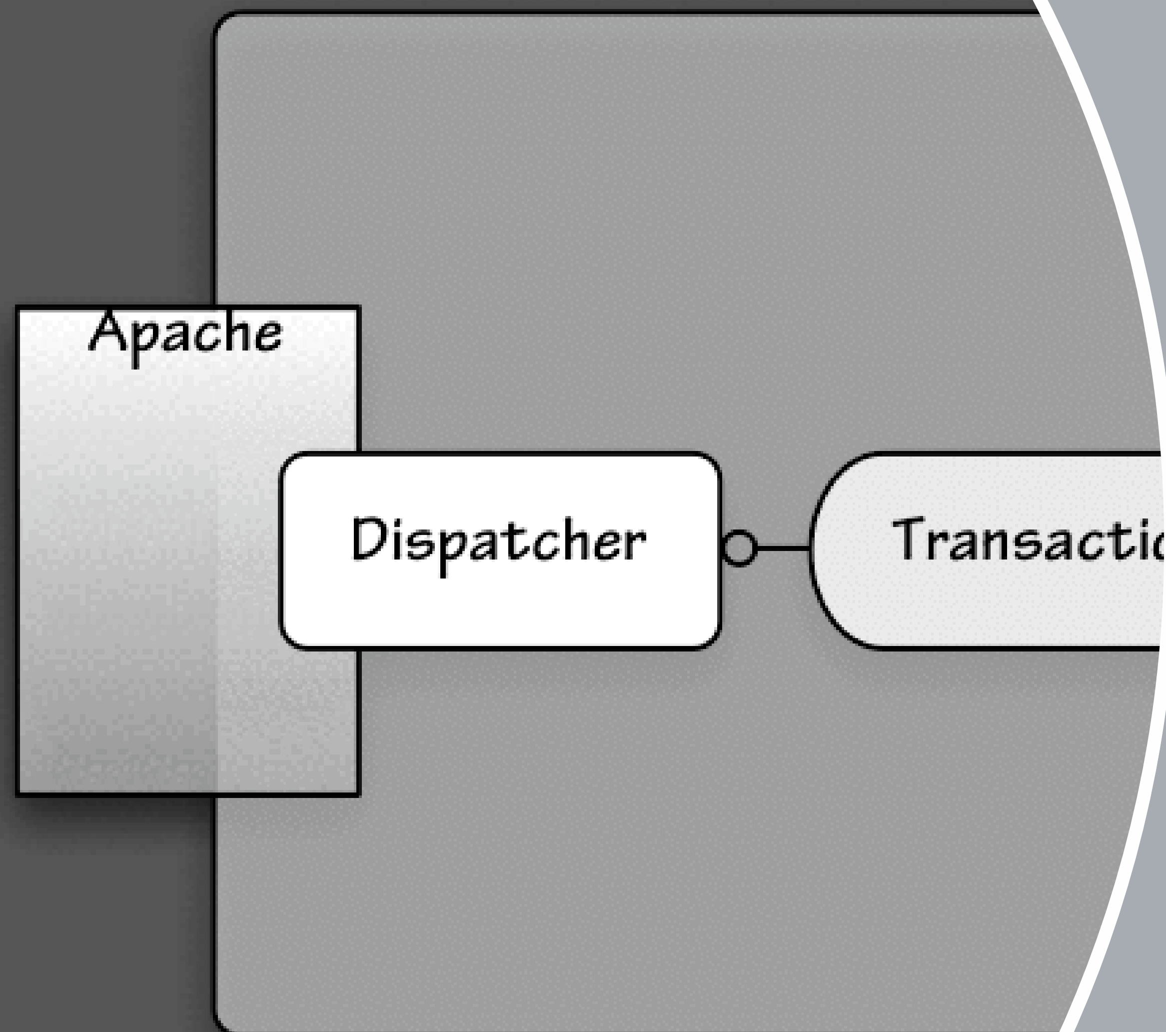
Architecture



Architecture

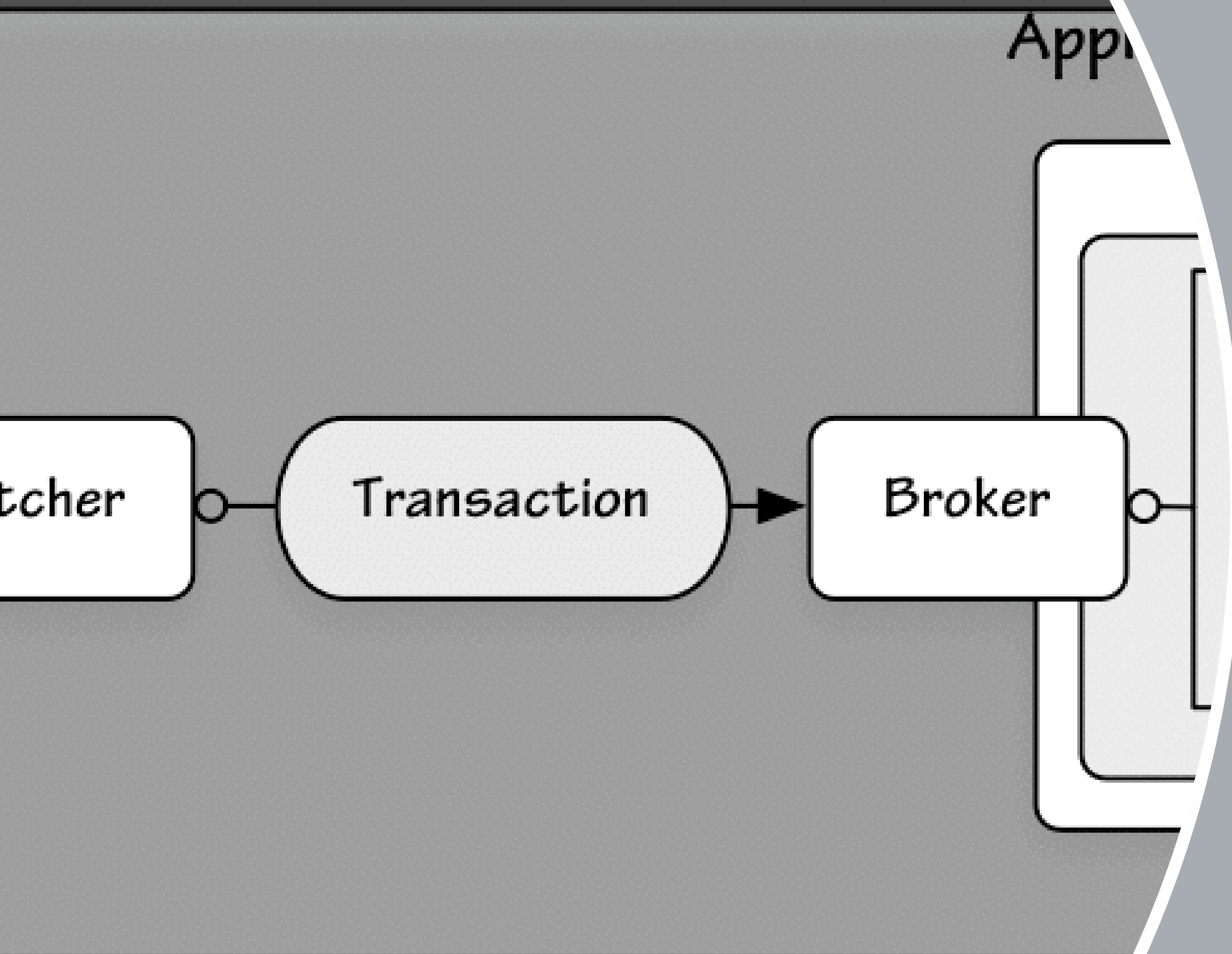
- Four Major Parts
  - Dispatcher
  - Transaction
  - Broker
  - Applets

Architecture: Dispatcher



- mod_ruby handler
- Maintains configuration object
- Wraps incoming Apache::Request in a Transaction object
- Passes Transaction to the Broker
- Handles rendering output and headers on the way back out

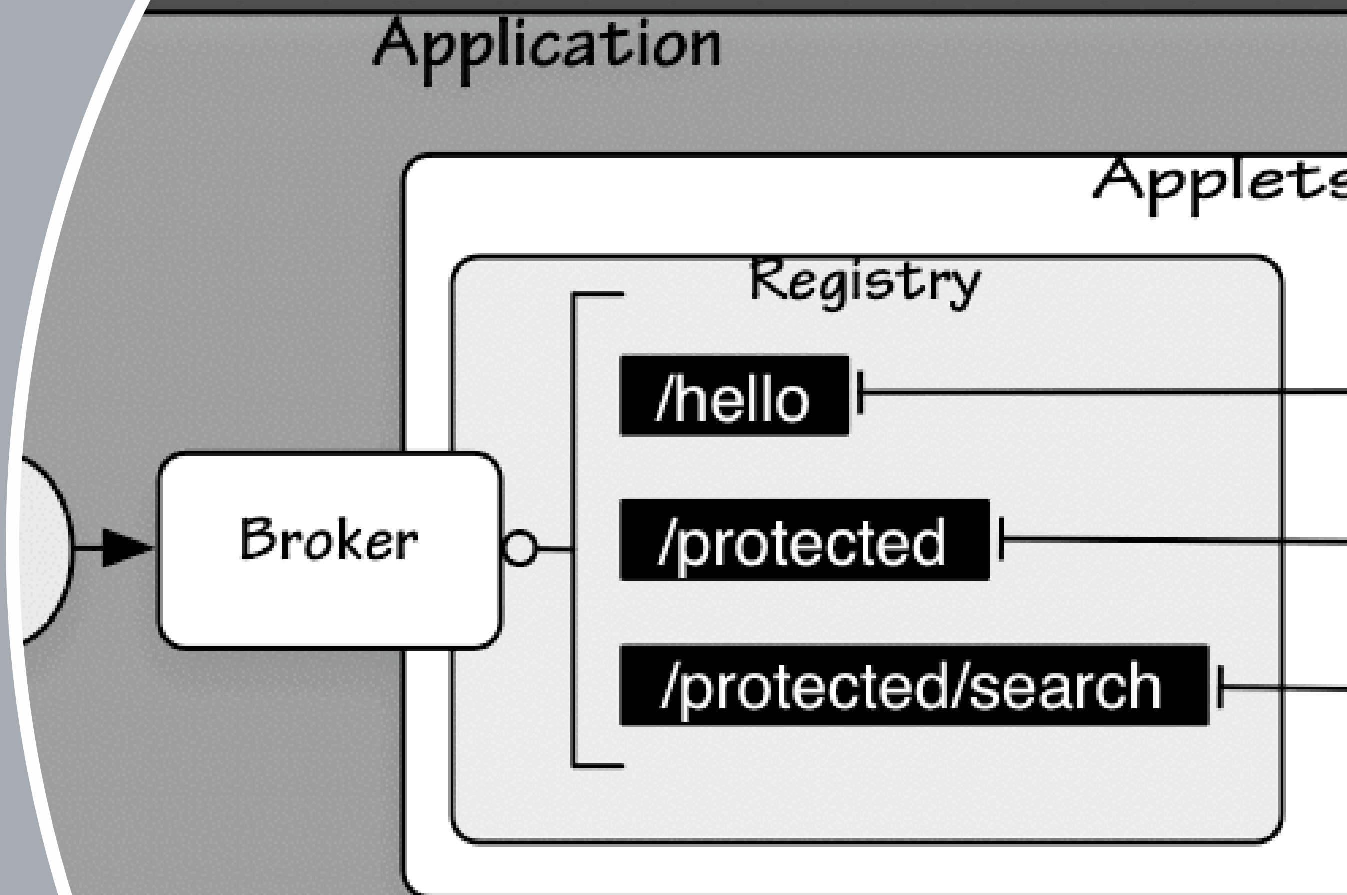
Architecture: Transaction



- Created by the dispatcher for every request
- Wraps (and delegates to) an Apache::Request
- Tracks transaction history, session, validated arguments
- Can be used to pass data between applets

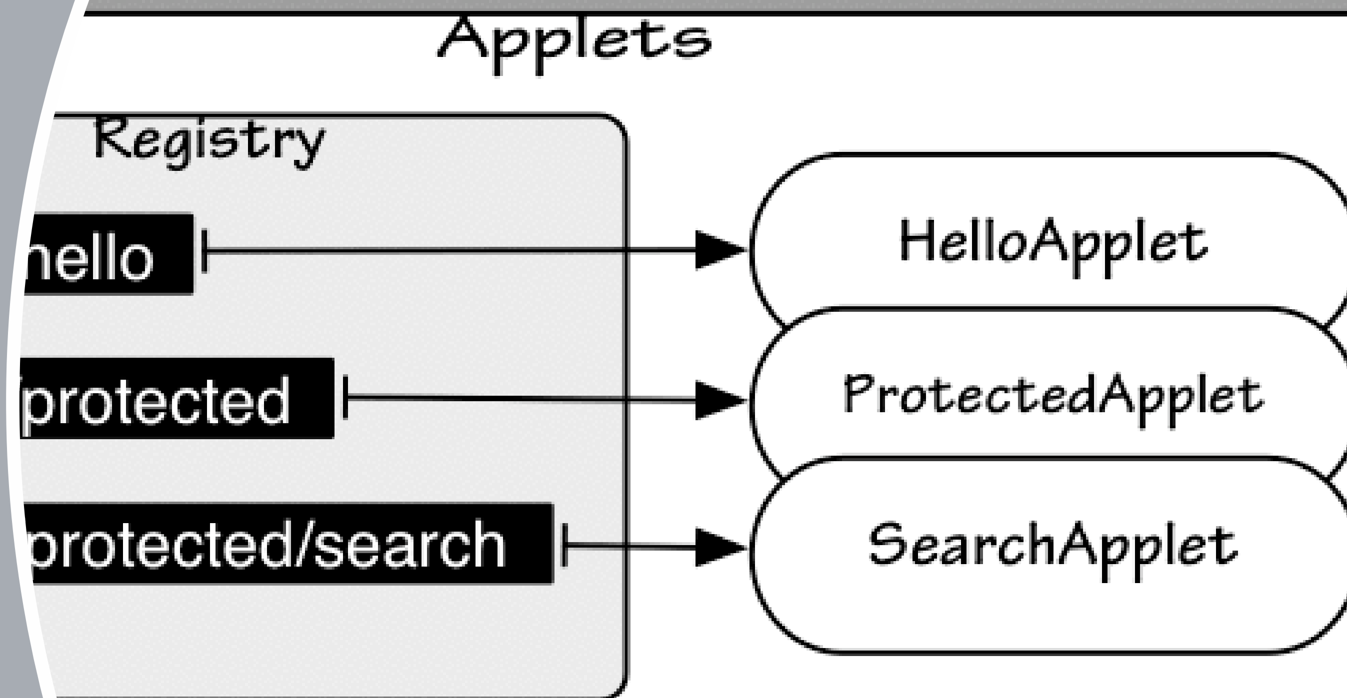
Architecture: Broker

- Manages loading, reloading, and mapping of applets to the application's URIspace.
- Decides how Transactions are handled, passing it to one or more applets
- Handles exceptions



Architecture: Applets

- Handles user requests with the action specified in the URI
- Methods provided in the base class for loading templates and validating arguments
- Sends content to the client by returning it
- Can be combined to run in sequence via delegation (chaining)

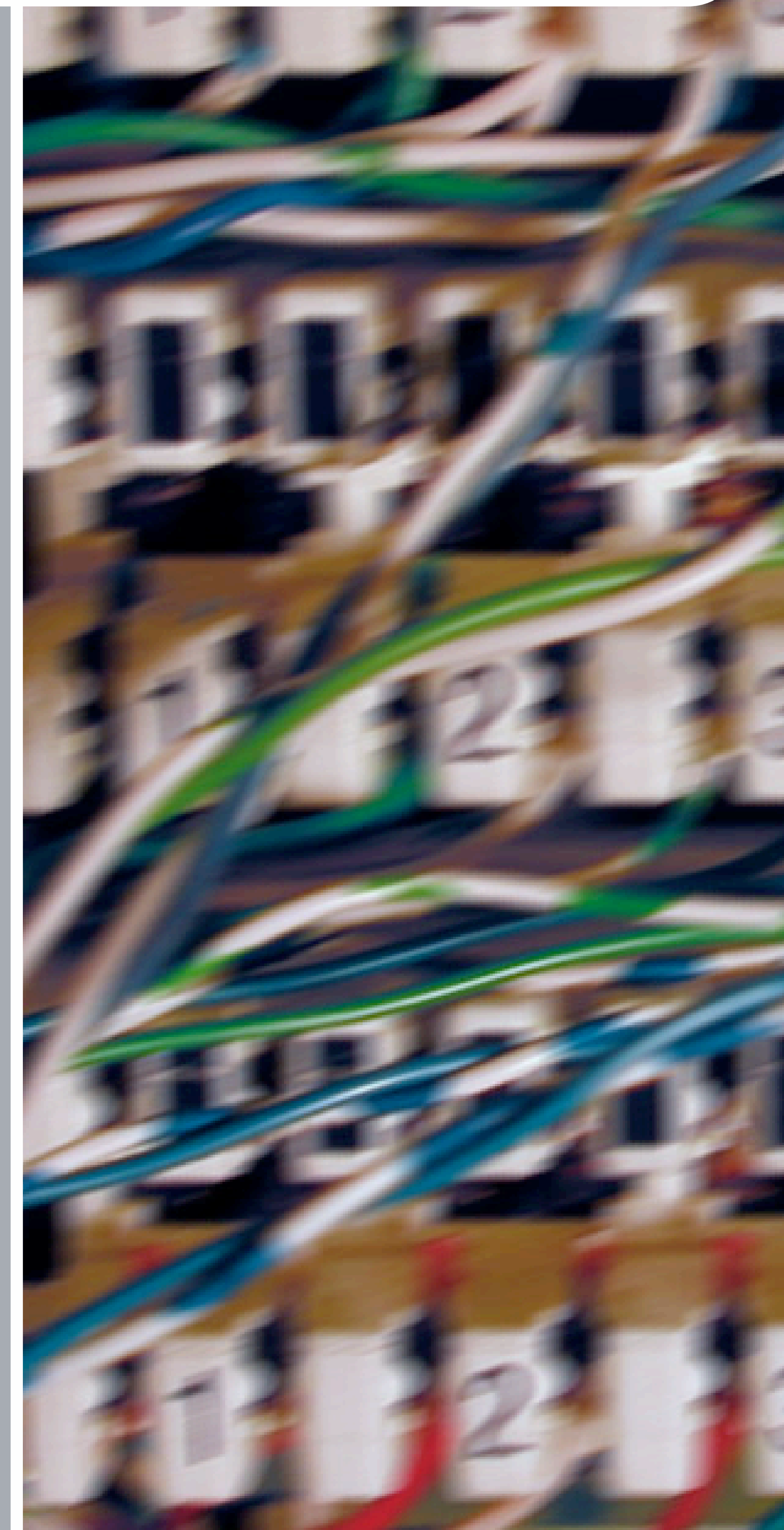


Tutorial

- Applets
- Templates
- Sessions
- Argument validation

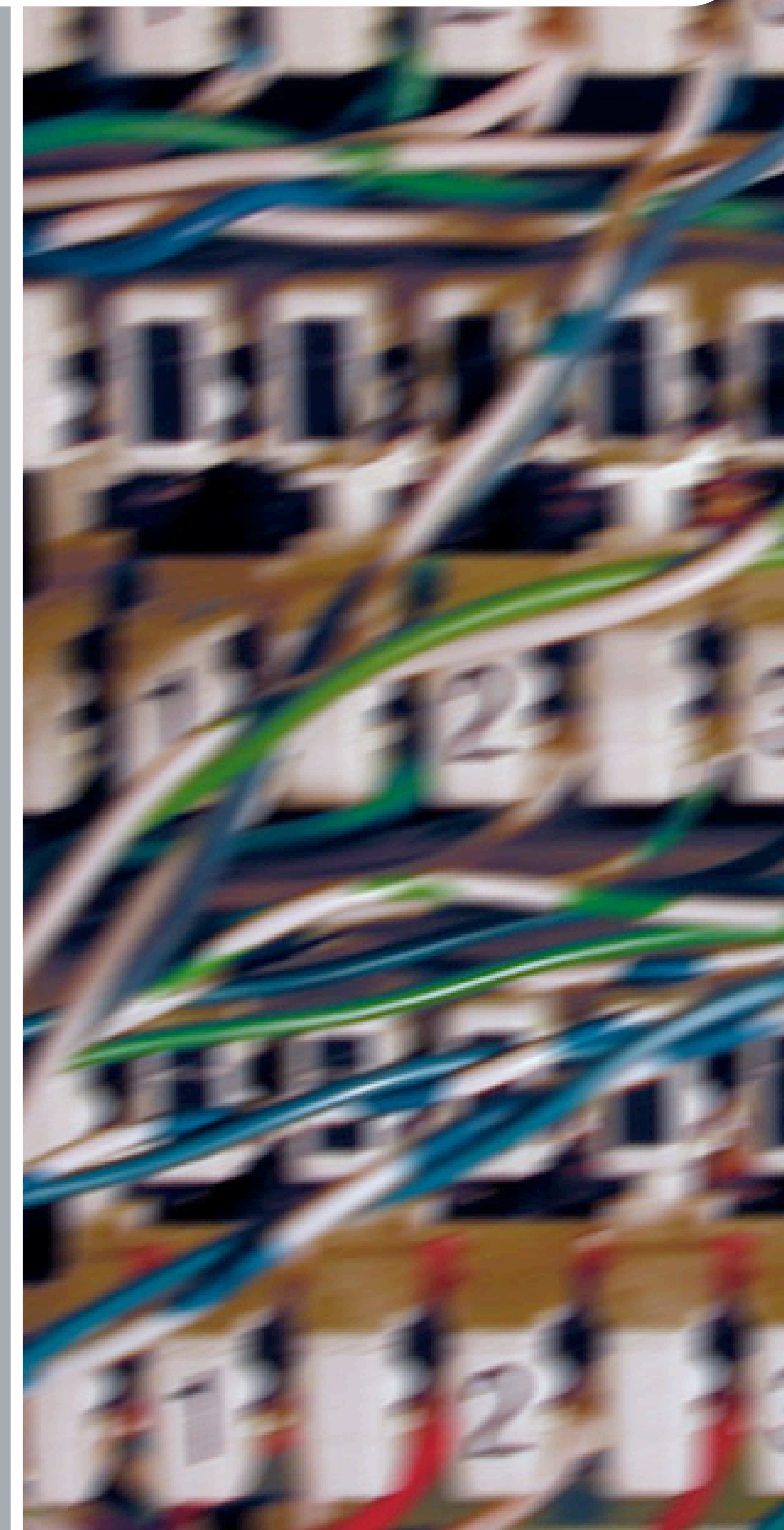
Tutorial: Applets

- Applets are made up of four parts
  - Signature
  - Actions
  - Delegator method (optional)
  - All the other gooey insides



Tutorial: Applets (cont.)

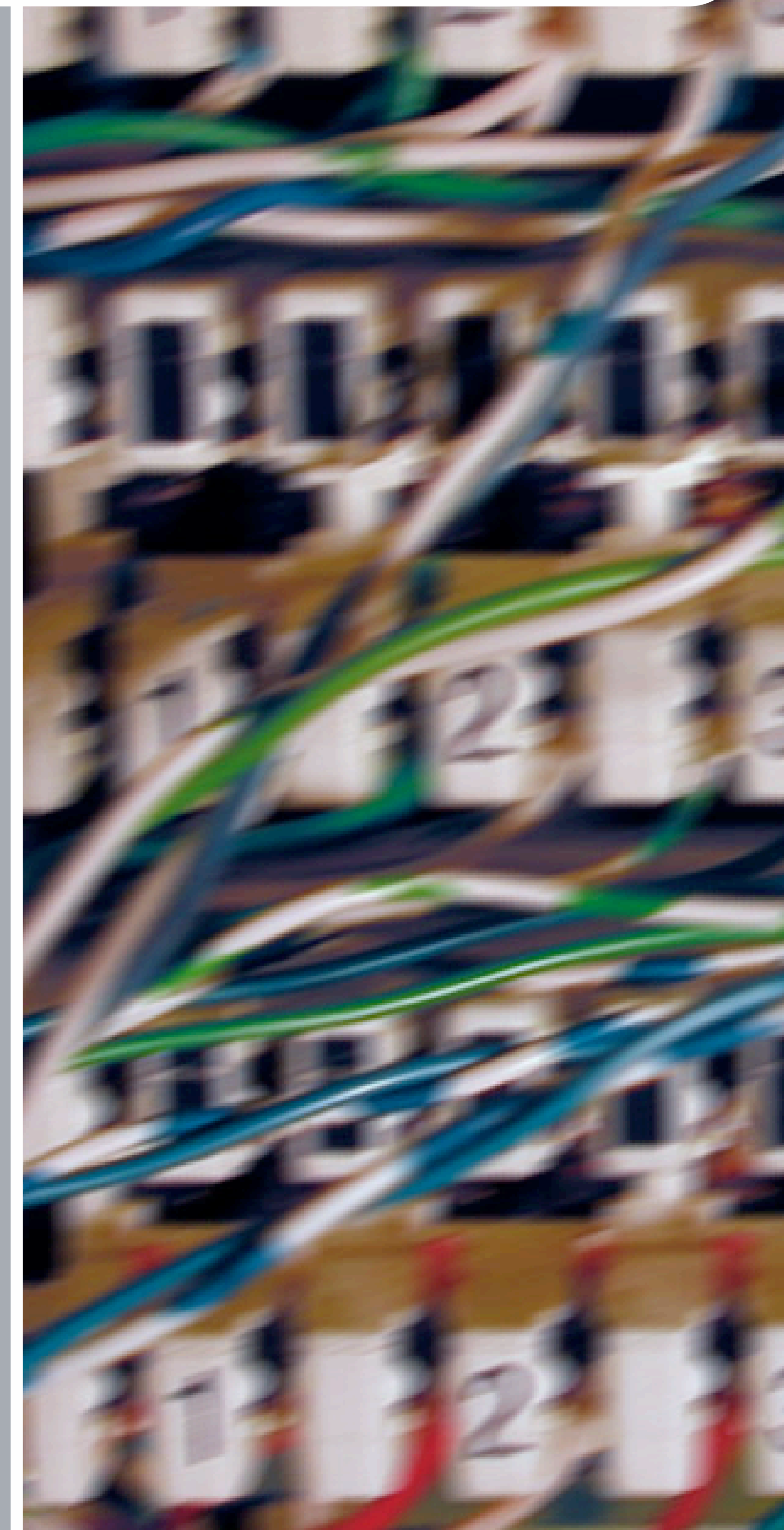
- Applet Signature
  - Defines metadata about the applet
  - Configures the template loader and argument validator
  - Can be a constant of the applet class called either `Signature` or `SIGNATURE`, or can also be a class attribute called `@signature`
  - Any omitted parts will be replaced with reasonable defaults



Tutorial: Applets (cont.)

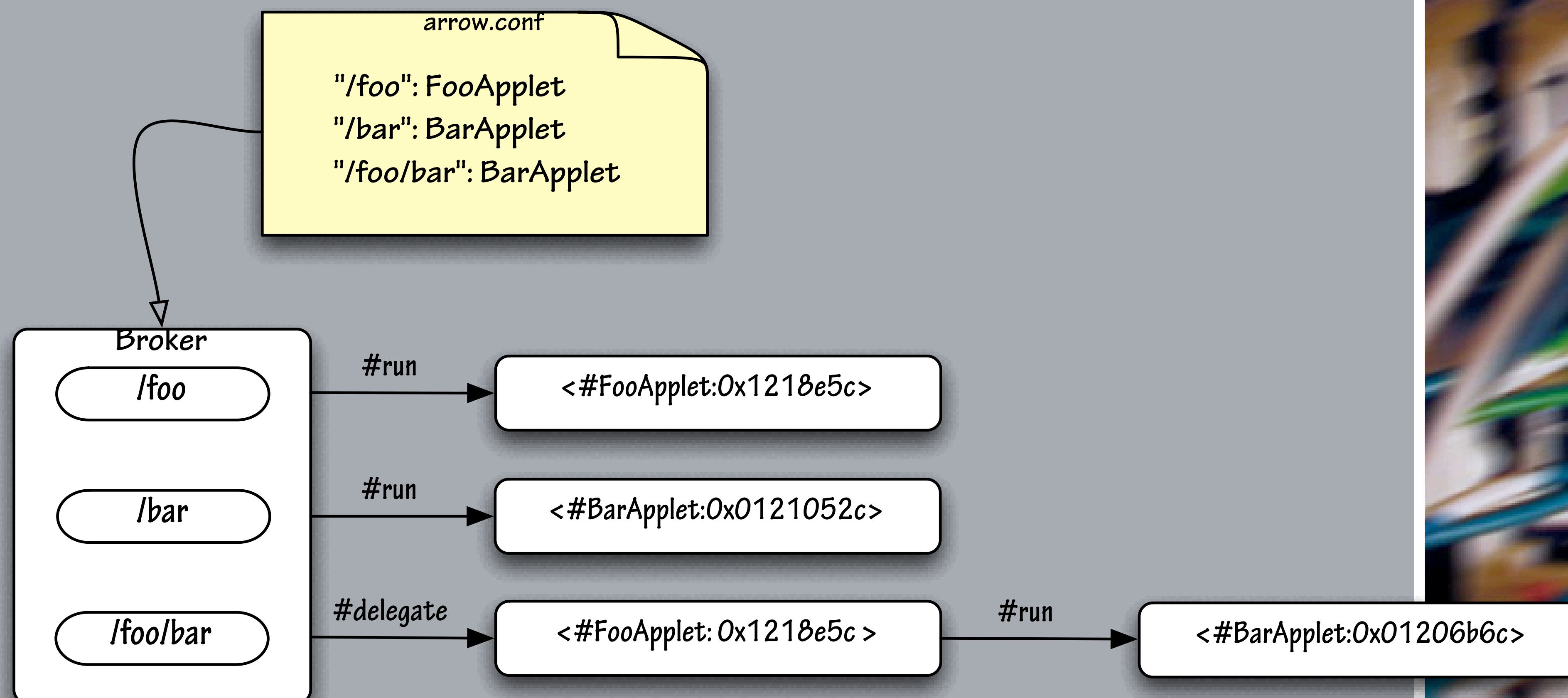
- An example signature:

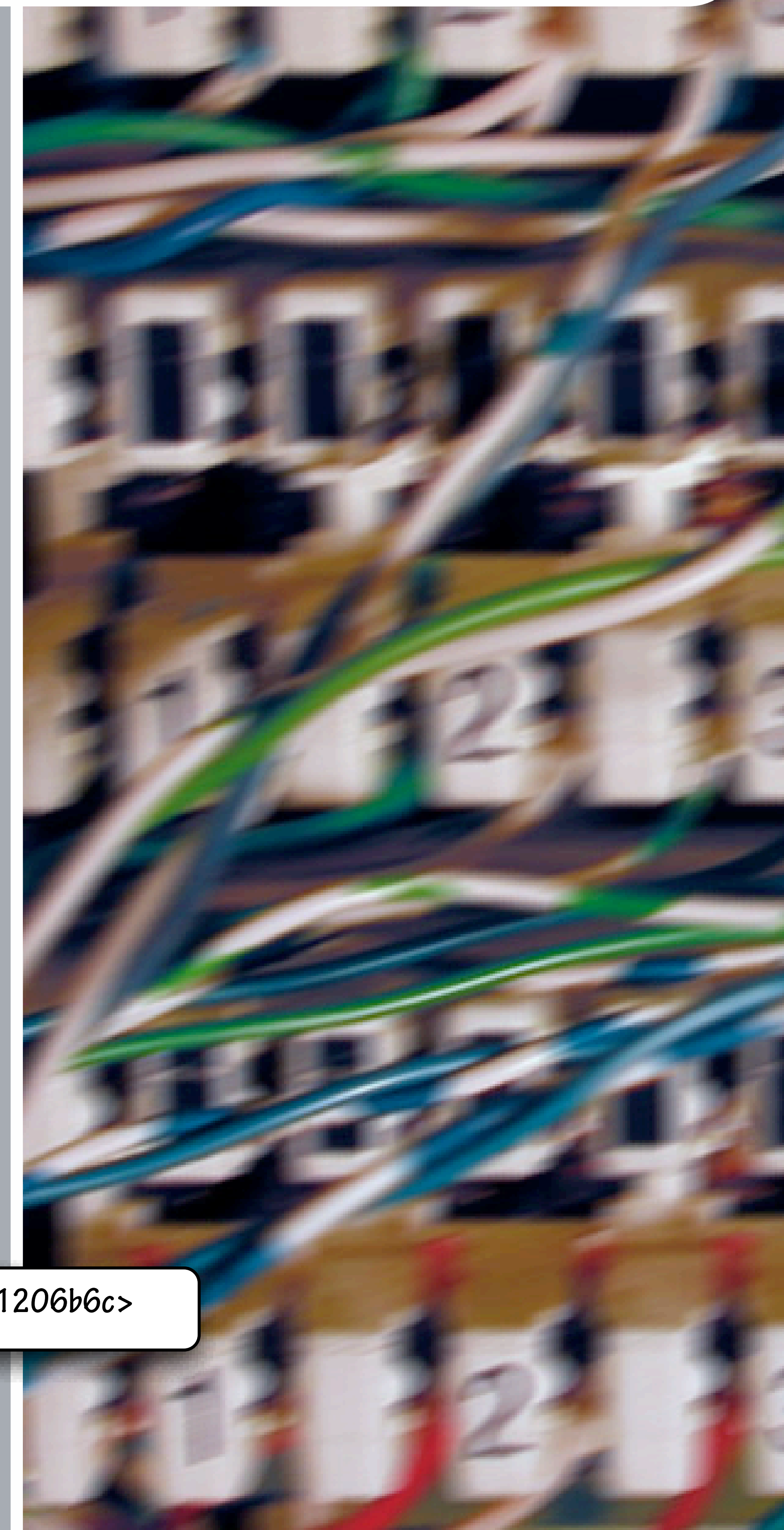
```
class HelloApplet
 Signature = {
  :name => "Hello World",
  :description => %{A 'hello world' applet.},
  :maintainer => "ged@FaerieMUD.org",
  :defaultAction => 'display',
  :templates => {
    :templated => 'hello-world.tmpl',
    :printsource => 'hello-world-src.tmpl',
  },
 }
 ...
end
```



Tutorial: Applets

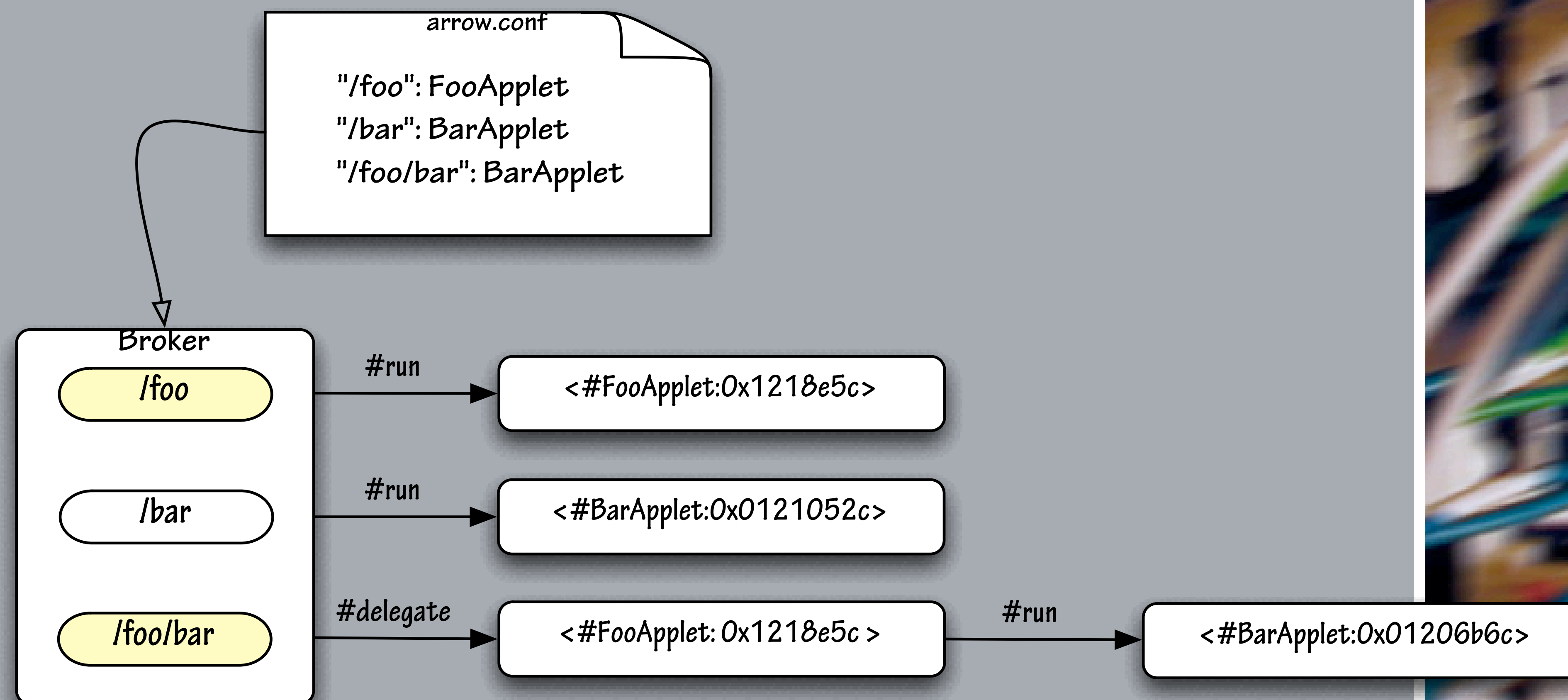
- Configuration maps URIs to applets when the Broker is loaded
- Request's URI is mapped onto one or more applets

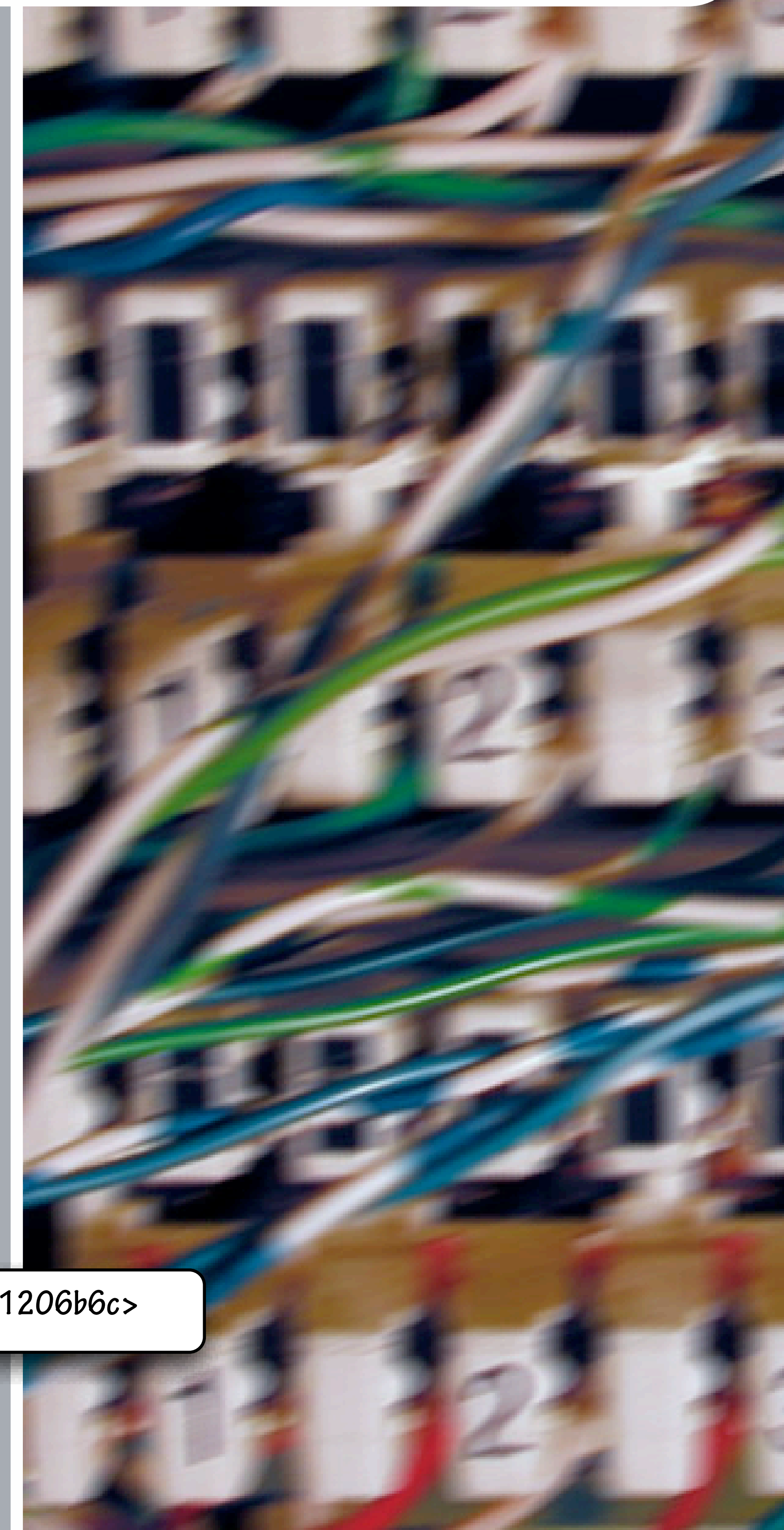




Tutorial: Applets (cont.)

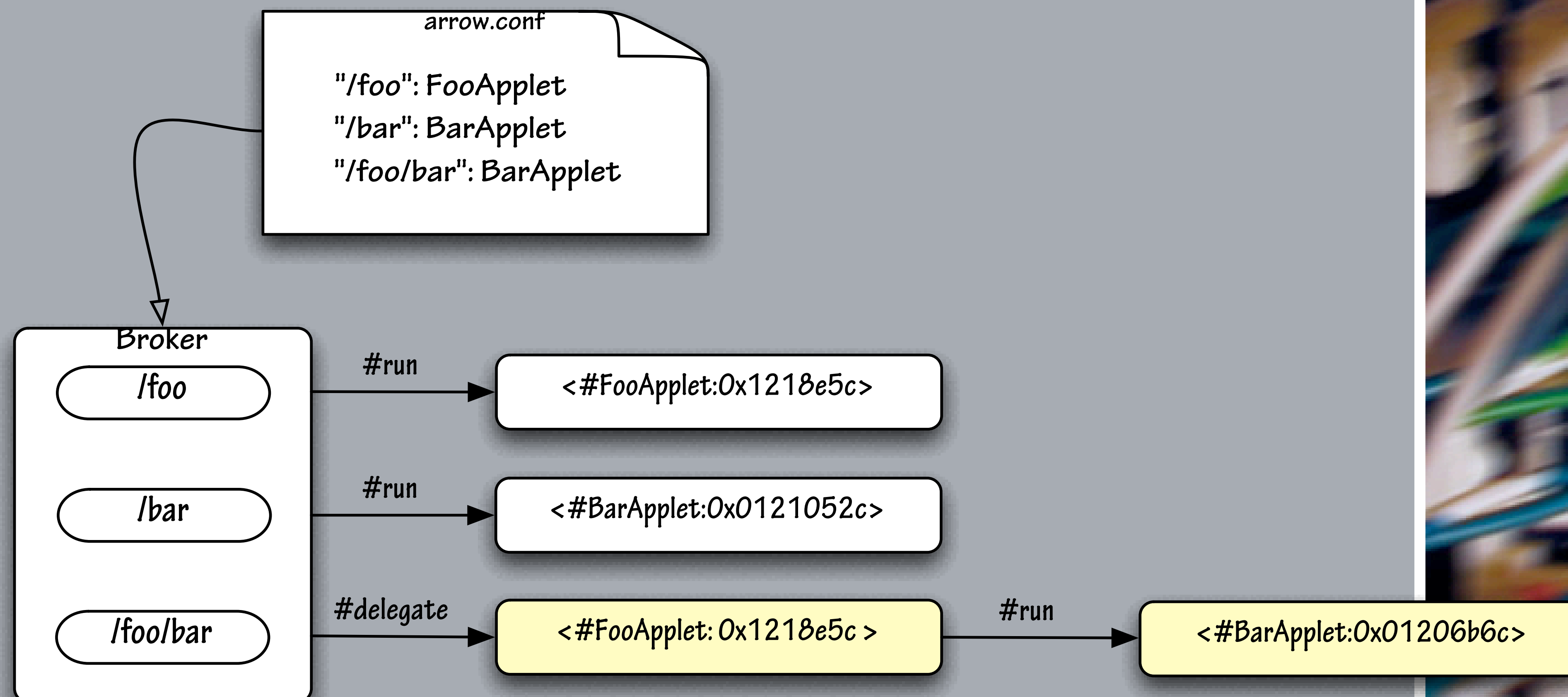
- Configuration maps URIs to applets when the Broker is loaded
- Request's URI is mapped onto one or more applets

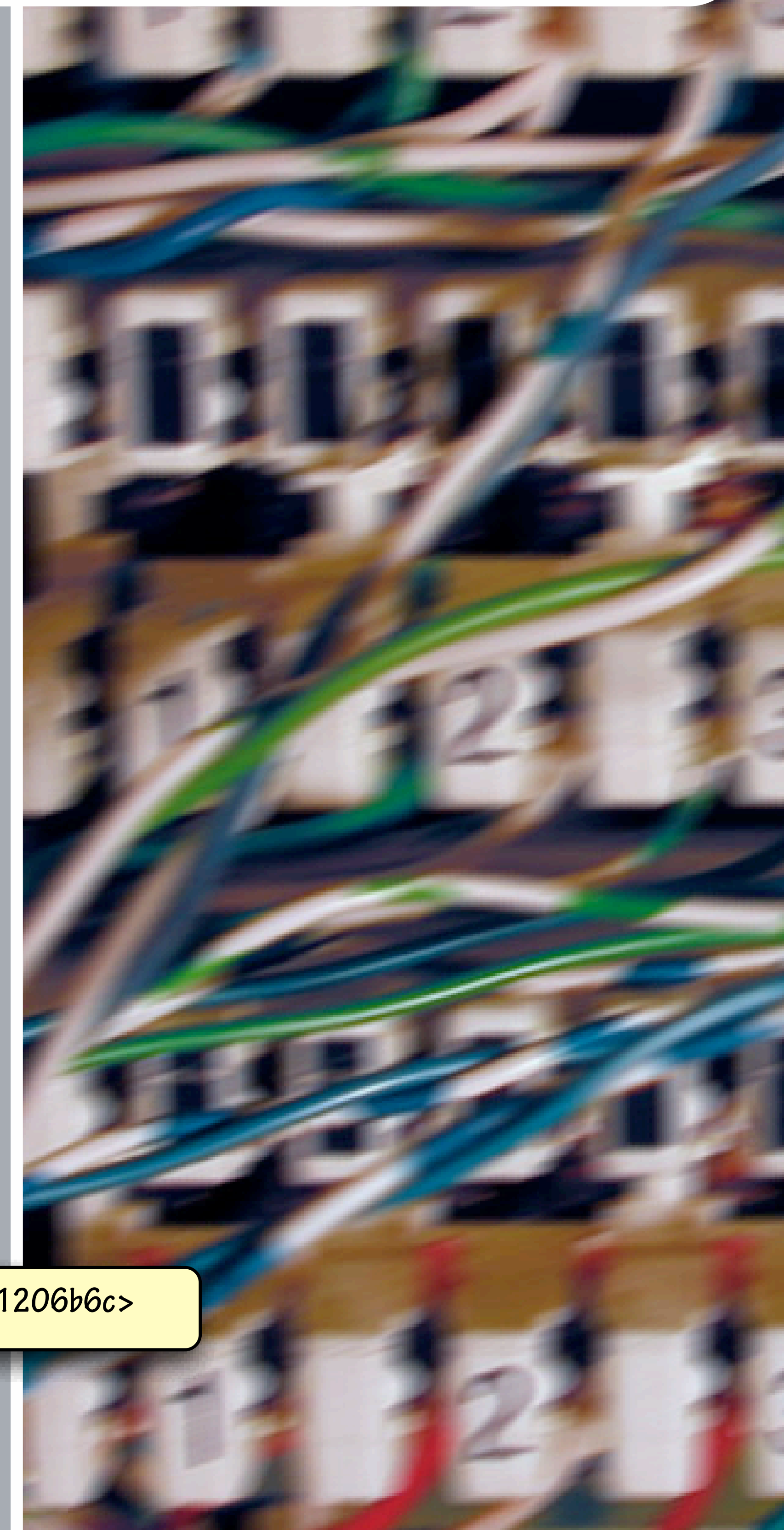




Tutorial: Applets (cont.)

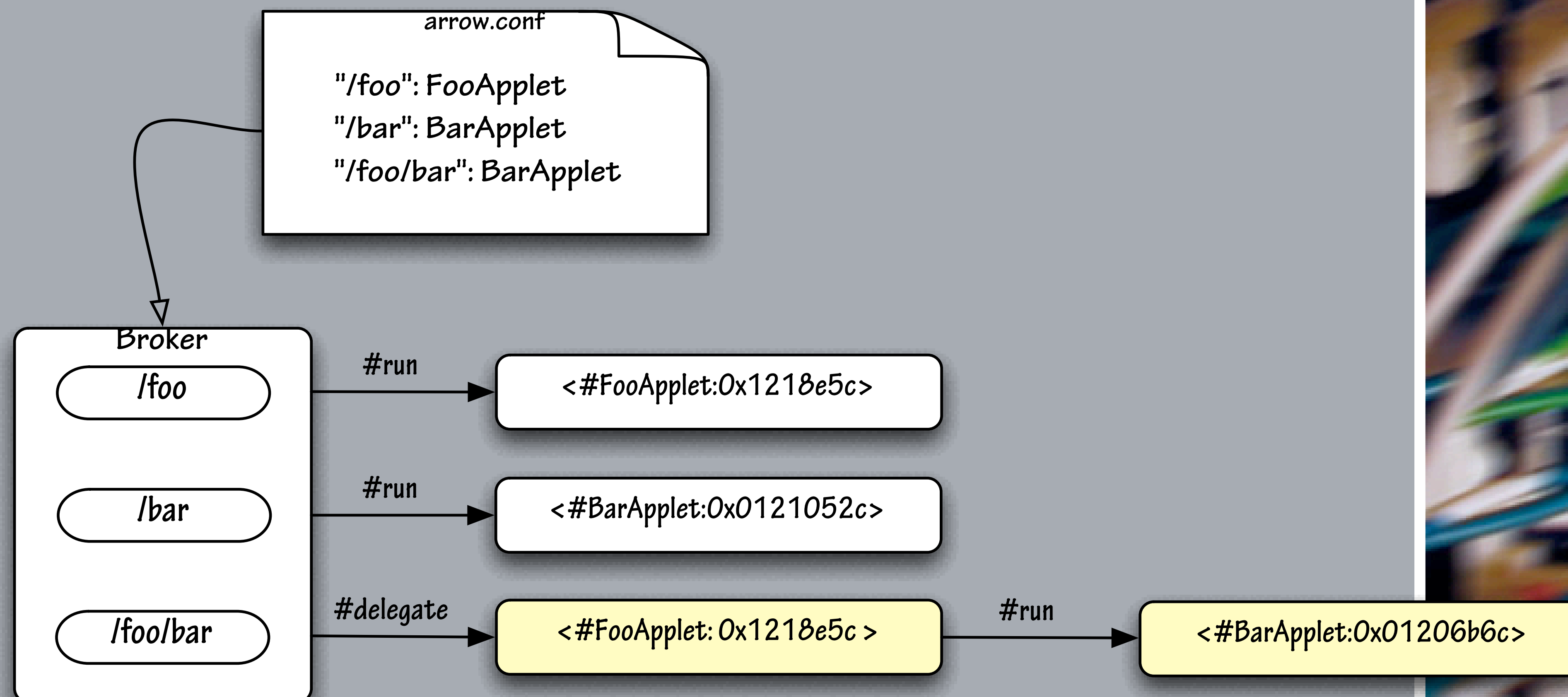
- If a URI matches more than one applet, all but the last are called via their `#delegate` method

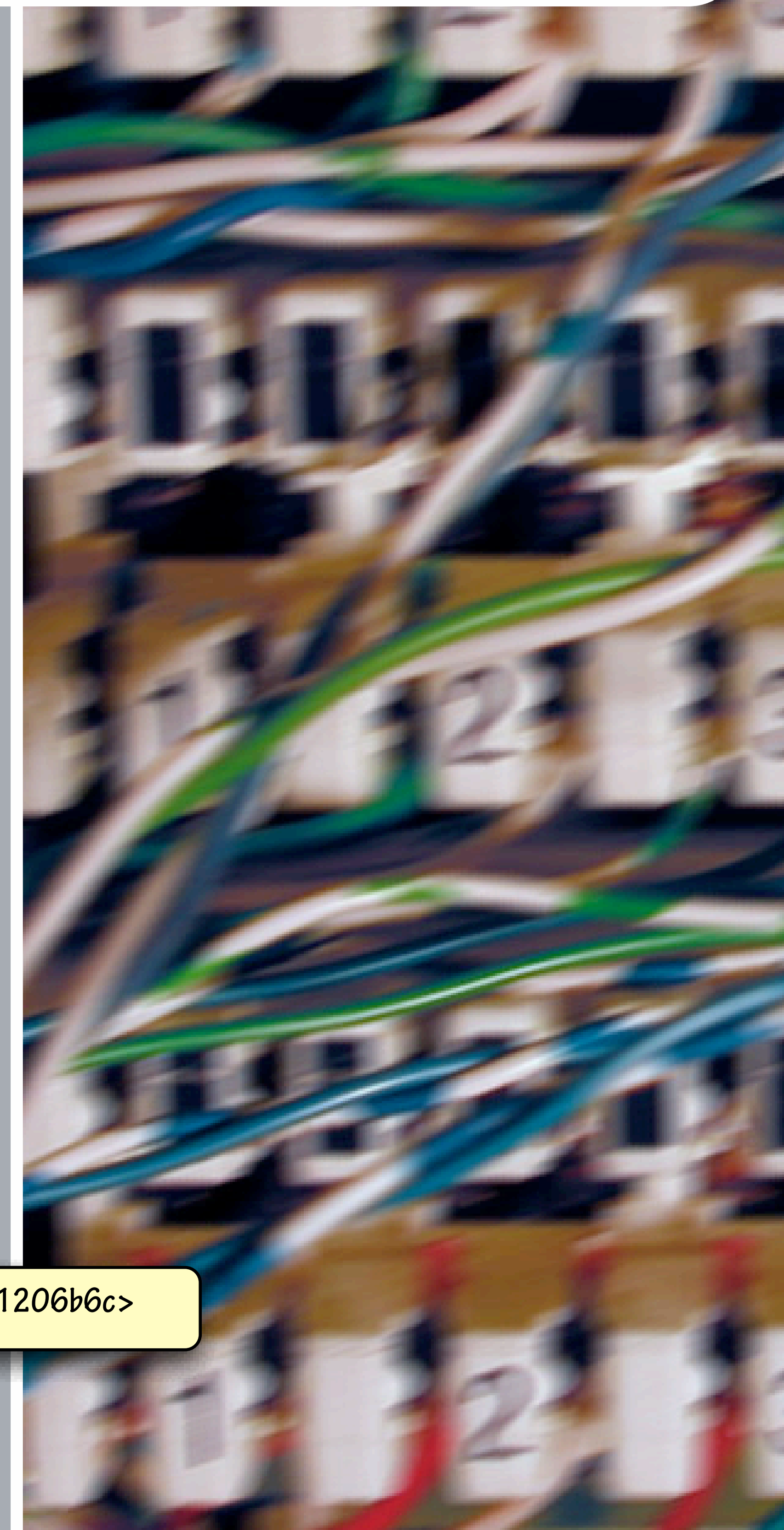




Tutorial: Applets (cont.)

- The last applet has its `#run` method invoked with the transaction and any remaining bits of the URI





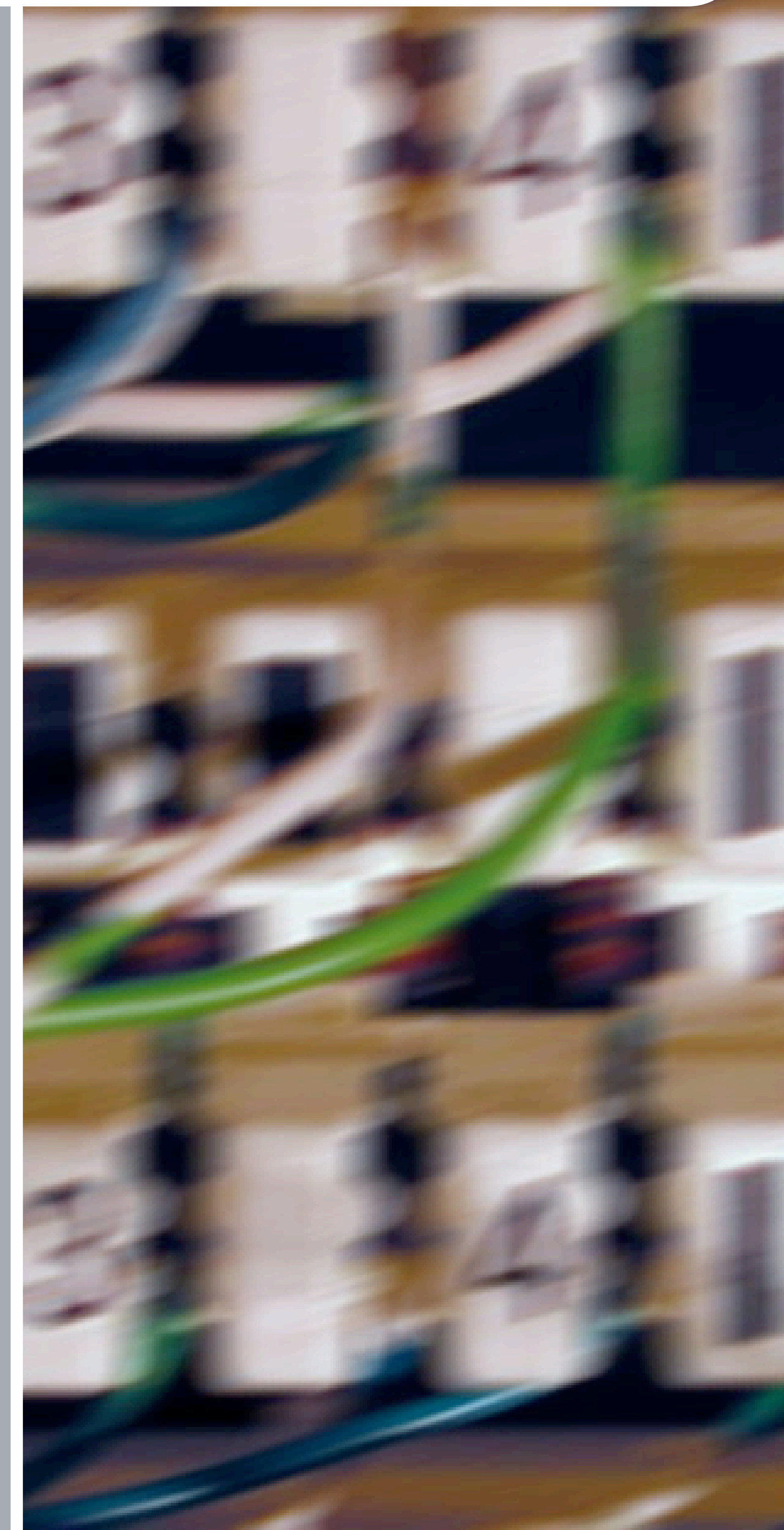
Tutorial: Applets (cont.)

- The part of the URI immediately after the applet name is the *action*, which maps to a method on the applet.

```
/staff/list ->
StaffApplet#list_action
```

- Action methods are methods with an `_action` suffix.

```
def hello_action( txn, *args )
  return “Hello”
end
```

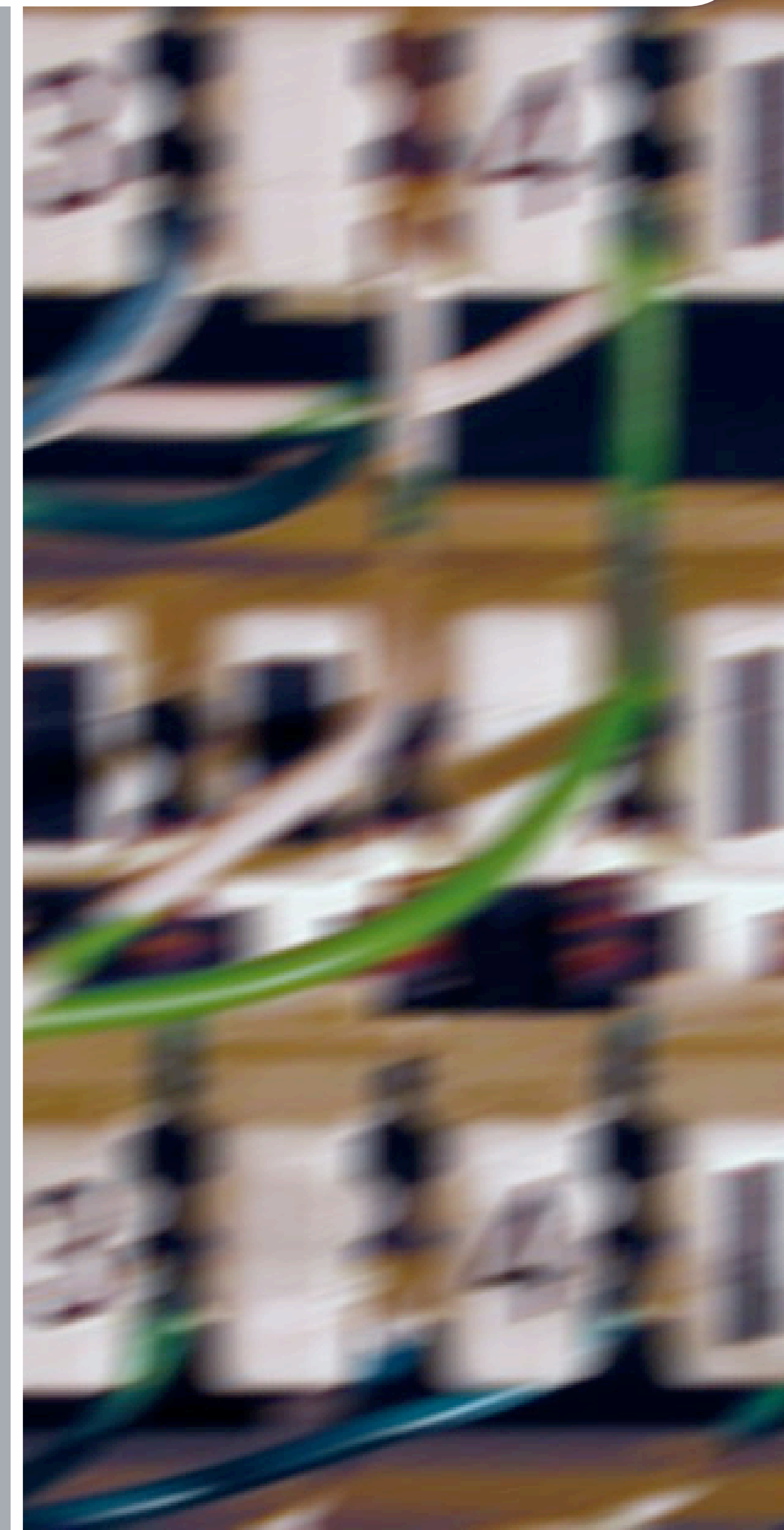


Tutorial: Applets (cont.)

- Or, as a shortcut, you can use the `action` declaration:

```
action( :display ) {|txn, *args|
  return "Hello"
}
```

- True-ish return values are sent as the body of the response
- False-ish return values (nil and false) cause the request to be declined by the Dispatcher.



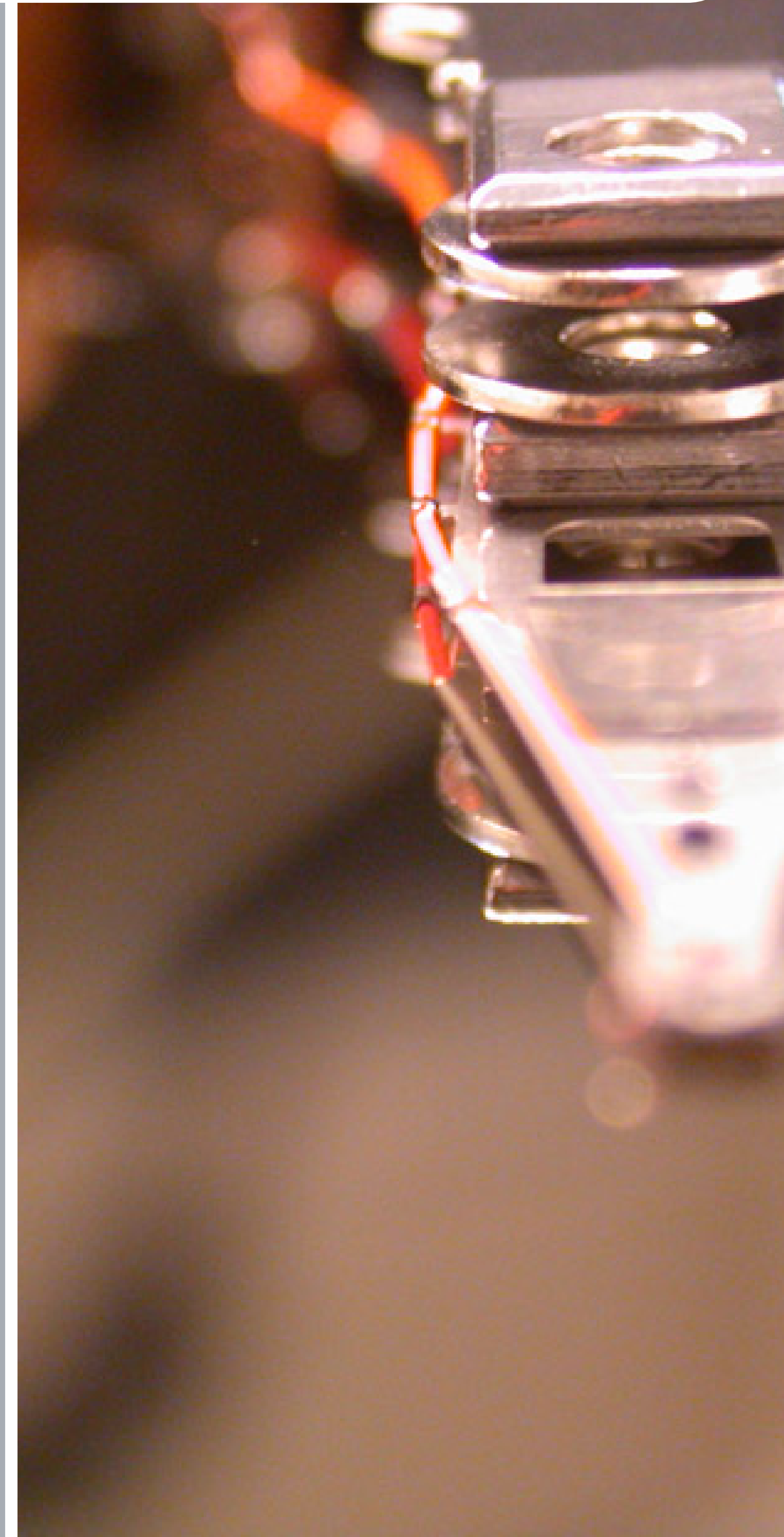
Tutorial: Applet Chaining

- Applets can also be aggregated by chaining them together.
- Applets called in the middle of a chain will have their `#delegate` method called; `yielding` from this method runs the next applet in the chain.

```
def delegate(txn, chain, *args)
  yield( chain )
end
```

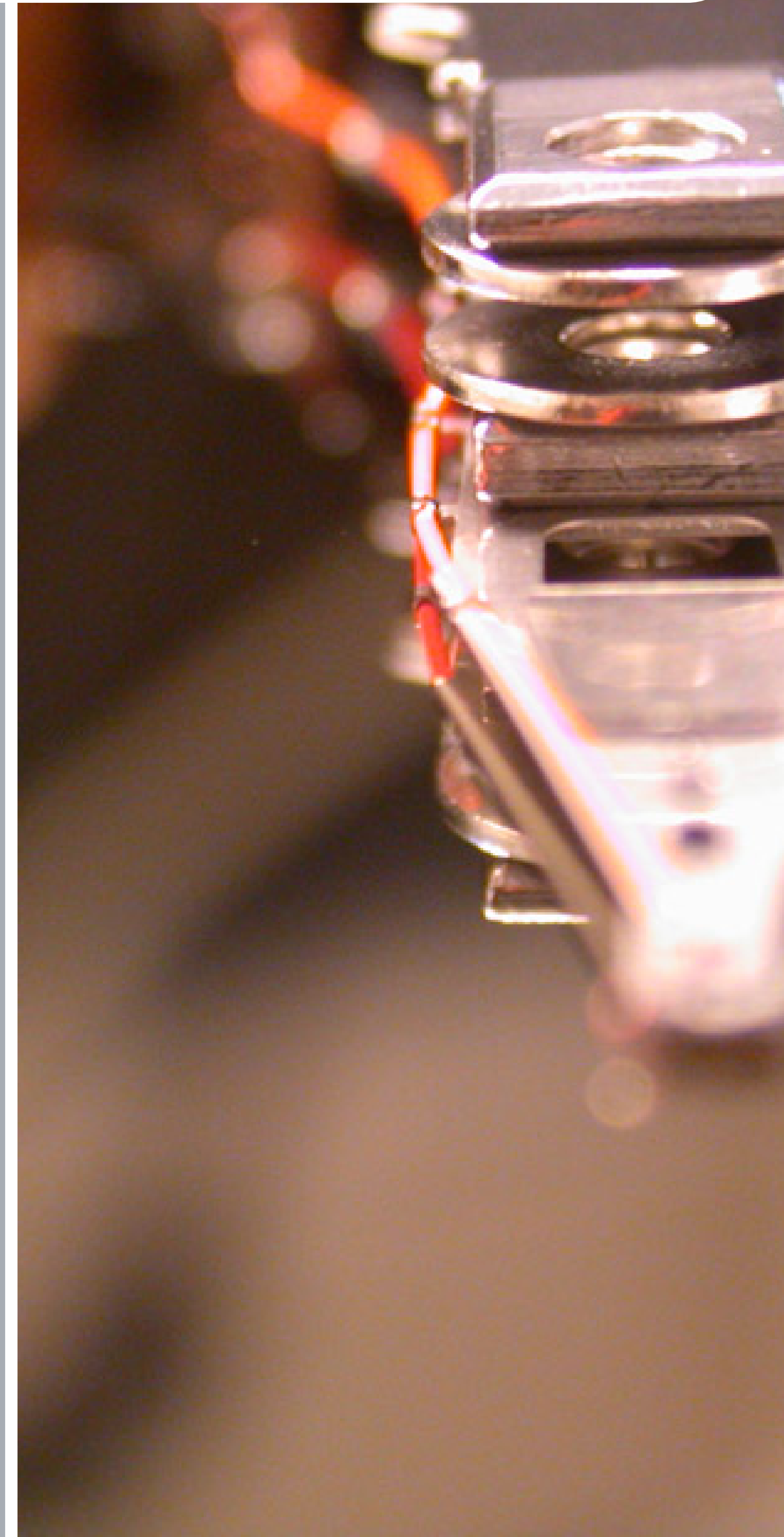
Tutorial: Templating

- Arrow has a built-in templating system
- If you want to use your own preferred one, you can.
- I'll only cover the built-in one today



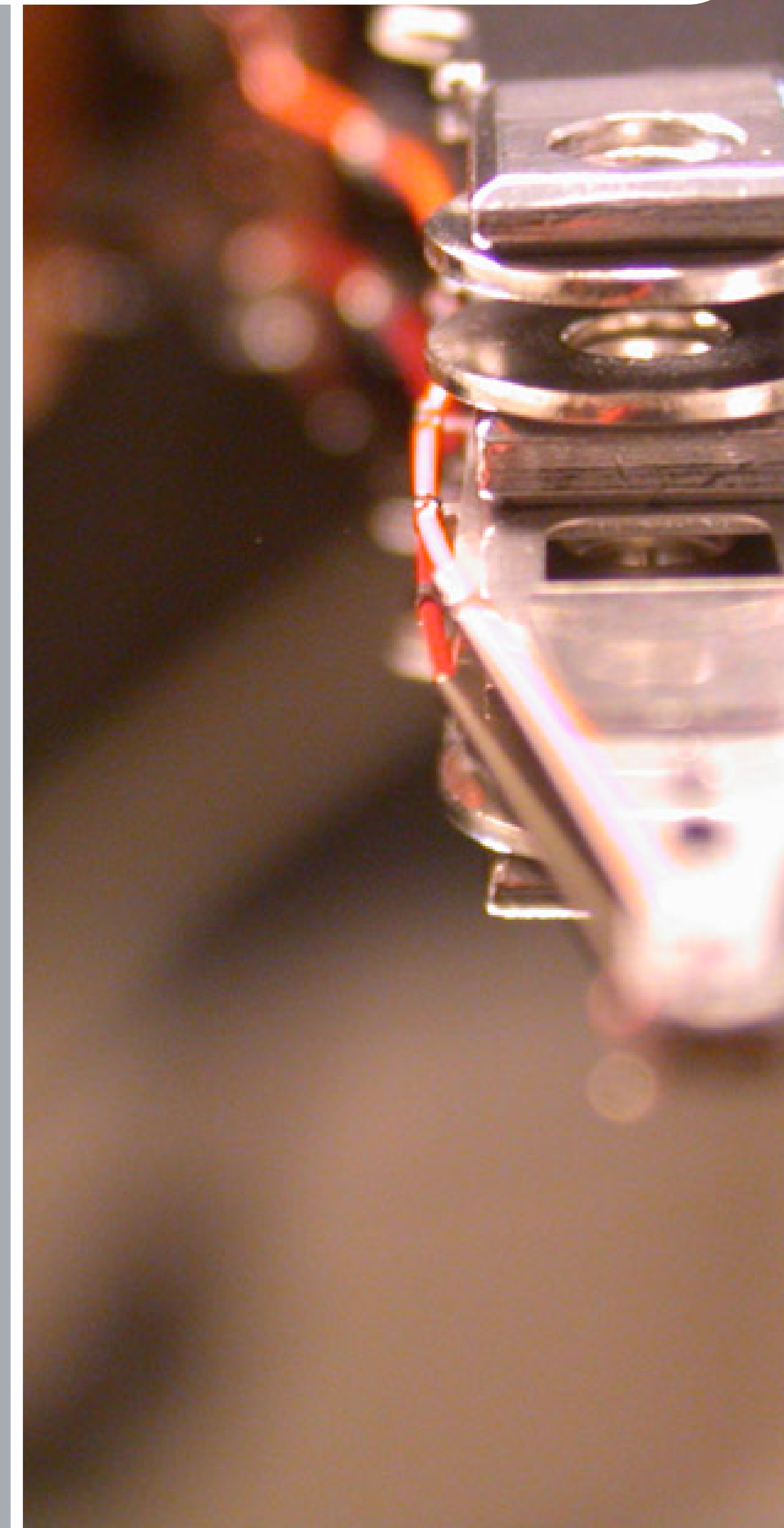
Tutorial: Templating (cont.)

- Templates are XHTML documents or document fragments
- Use XML processing-instructions to define directives (behavior)
- Templates are effectively class definitions, instances of which can be made to render desired output



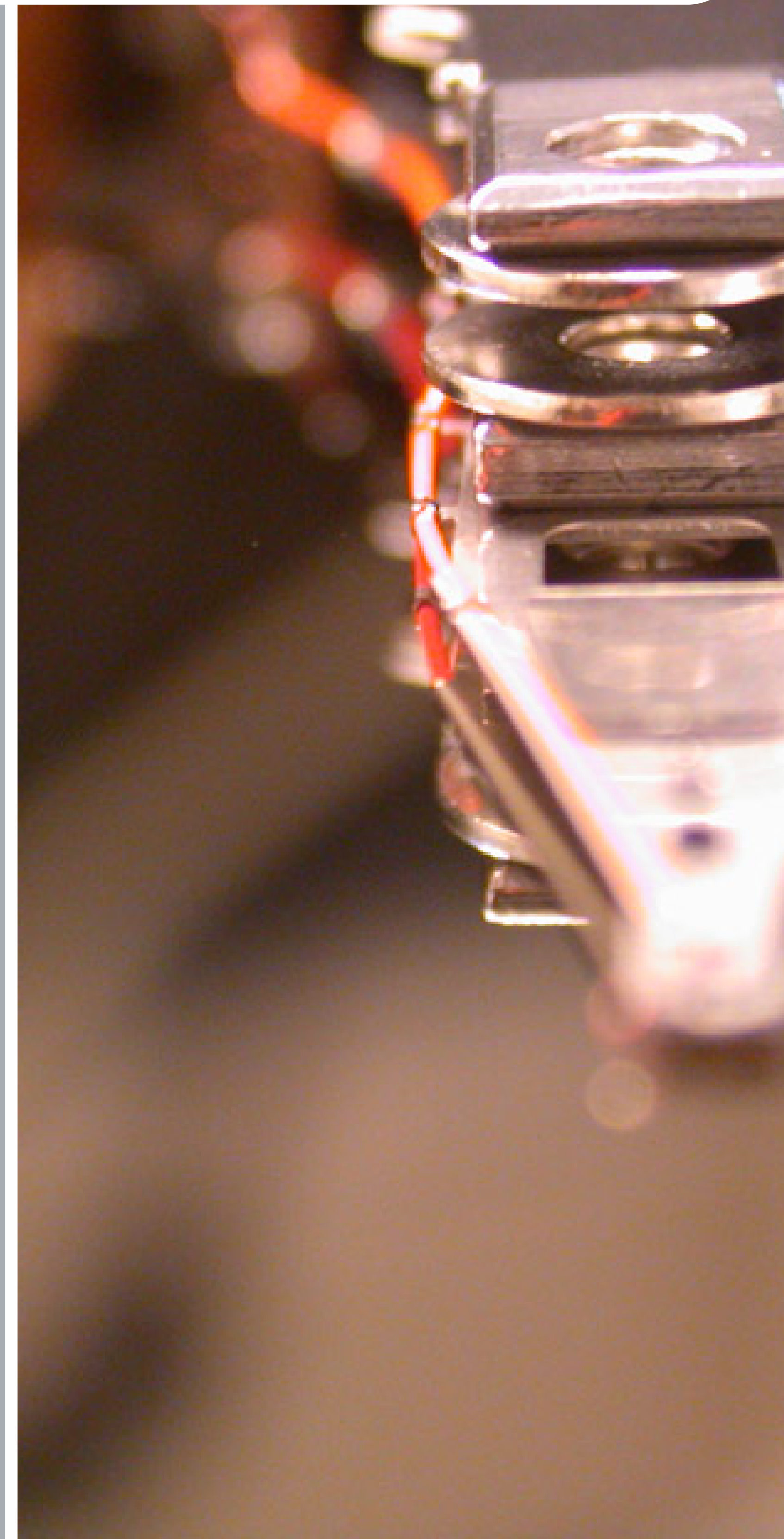
Tutorial: Templating (cont.)

- Square-bracket syntax for directives inside other tags: `[?...?]`
- Templates are infinitely nestable
- New directives can be added by dropping a file into a directory



Tutorial: Templating (cont.)

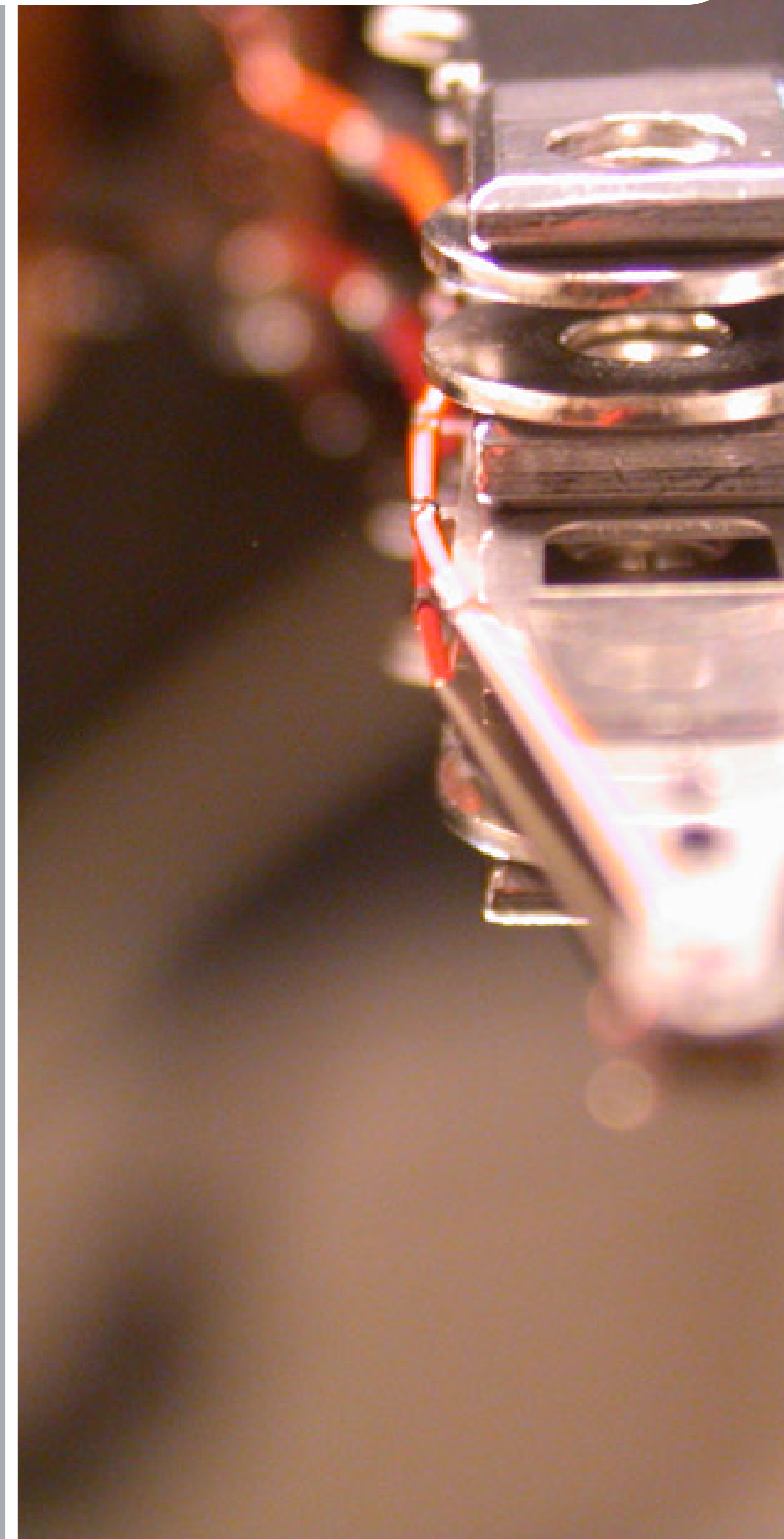
- The "target" part of the processing instruction defines the behavior of the method on the template object.
- The second part defines the name of the method
- E.g., `<?attr foo?>` is similar to `attr_accessor :foo`



Tutorial: Templating (cont.)

- An example template:

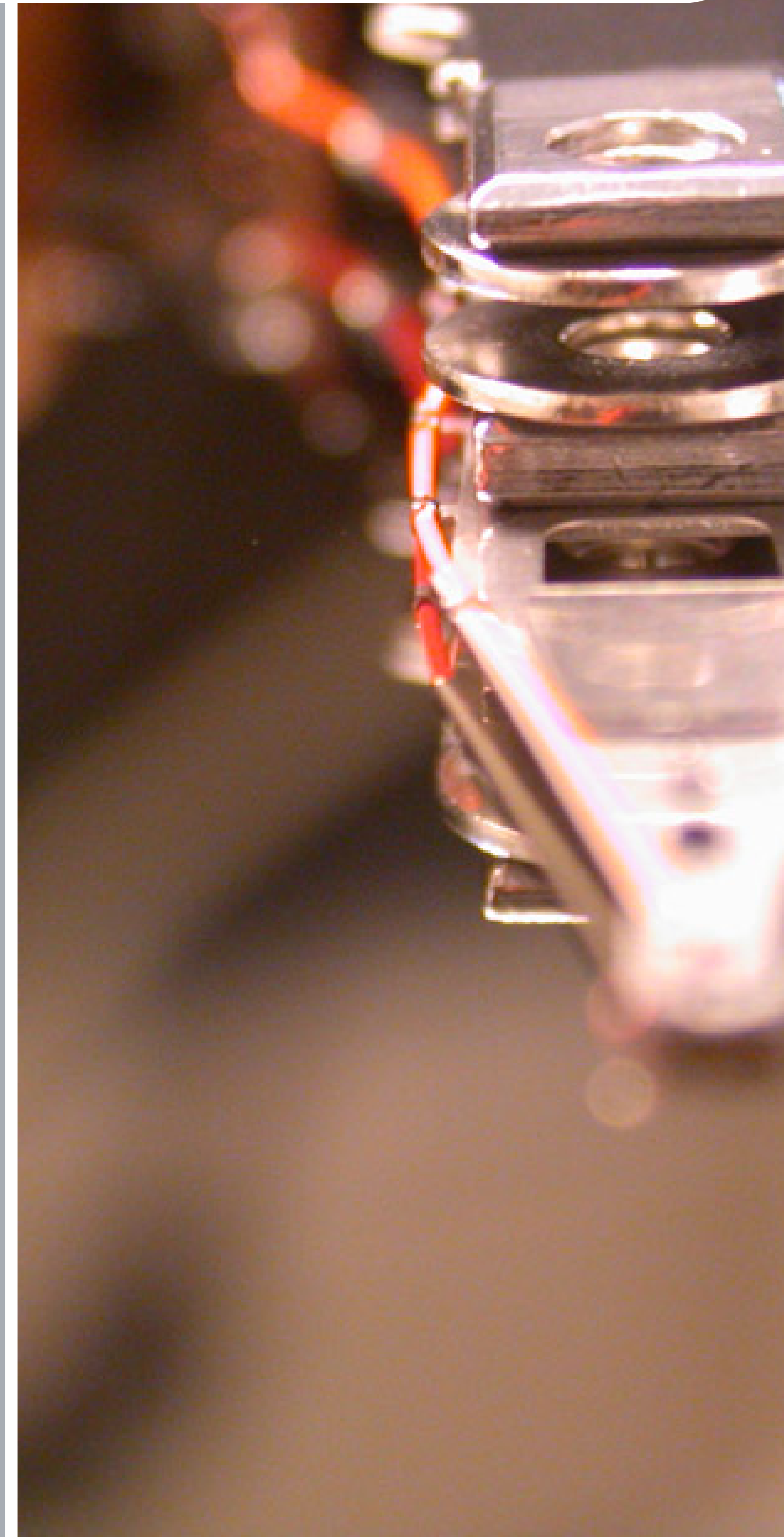
```
<!DOCTYPE html ...>
<html>
 <head>
  <title><?attr title?></title>
 </head>
 <body>
  <h1><?attr title?></h1>
  <?for para in paragraphs?>
    <p><?attr para?></p>
  <?end?>
 </body>
</html>
```



Tutorial: Templating (cont.)

- Which could be used (assuming `templ` is the template object):

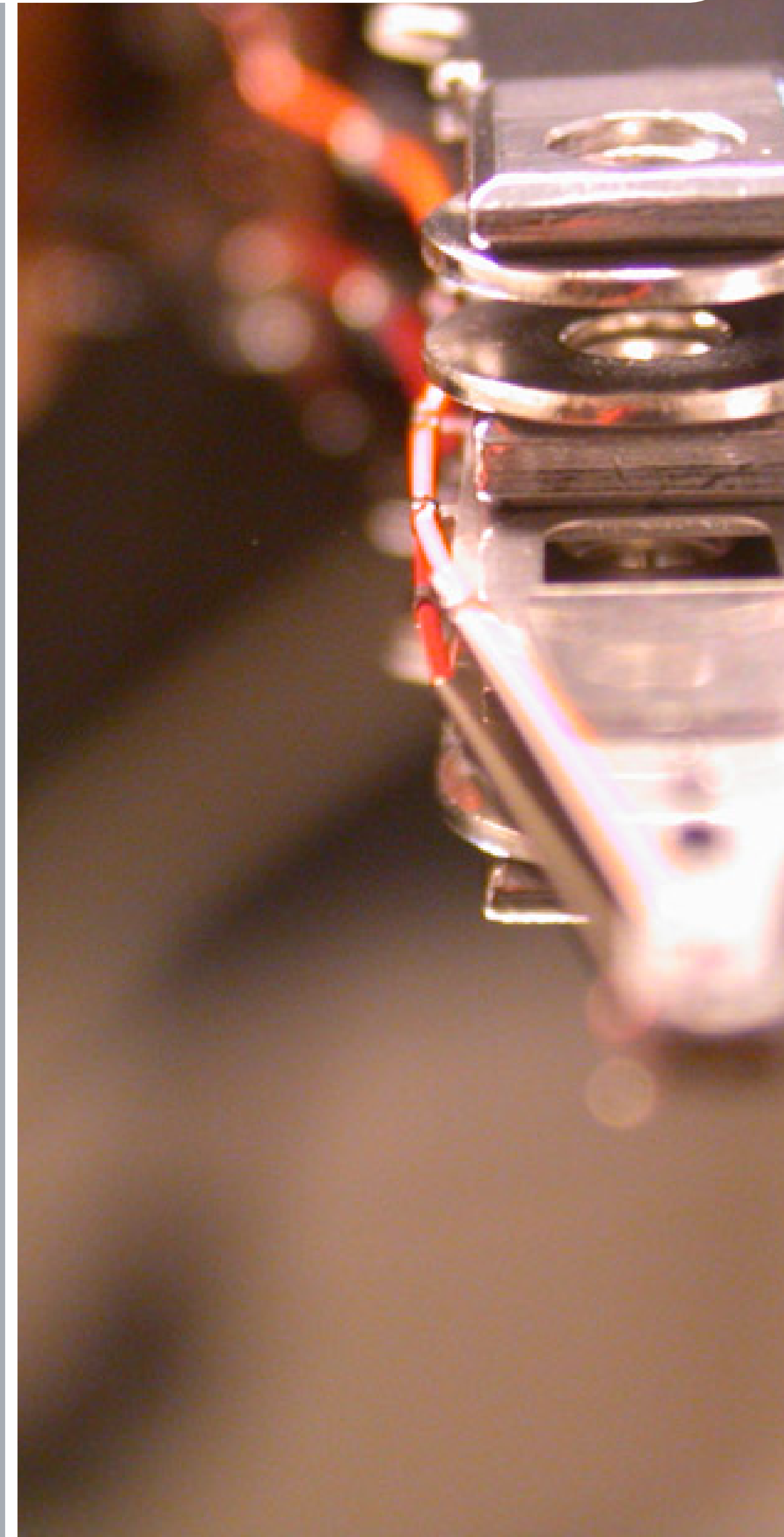
```
#!ruby

templ.title = “The Little Teapot”
templ.paragraphs <<
 “There once was a little teapot”+
 “ who lived all alone under a“+
 “ sycamore tree.”
templ.paragraphs <<
 “He was lonely, as one might”+
 “ expect of a teapot living on”+
 “ his own.“

return templ
```



Tutorial: Templating (cont.)

- Which would render thusly:

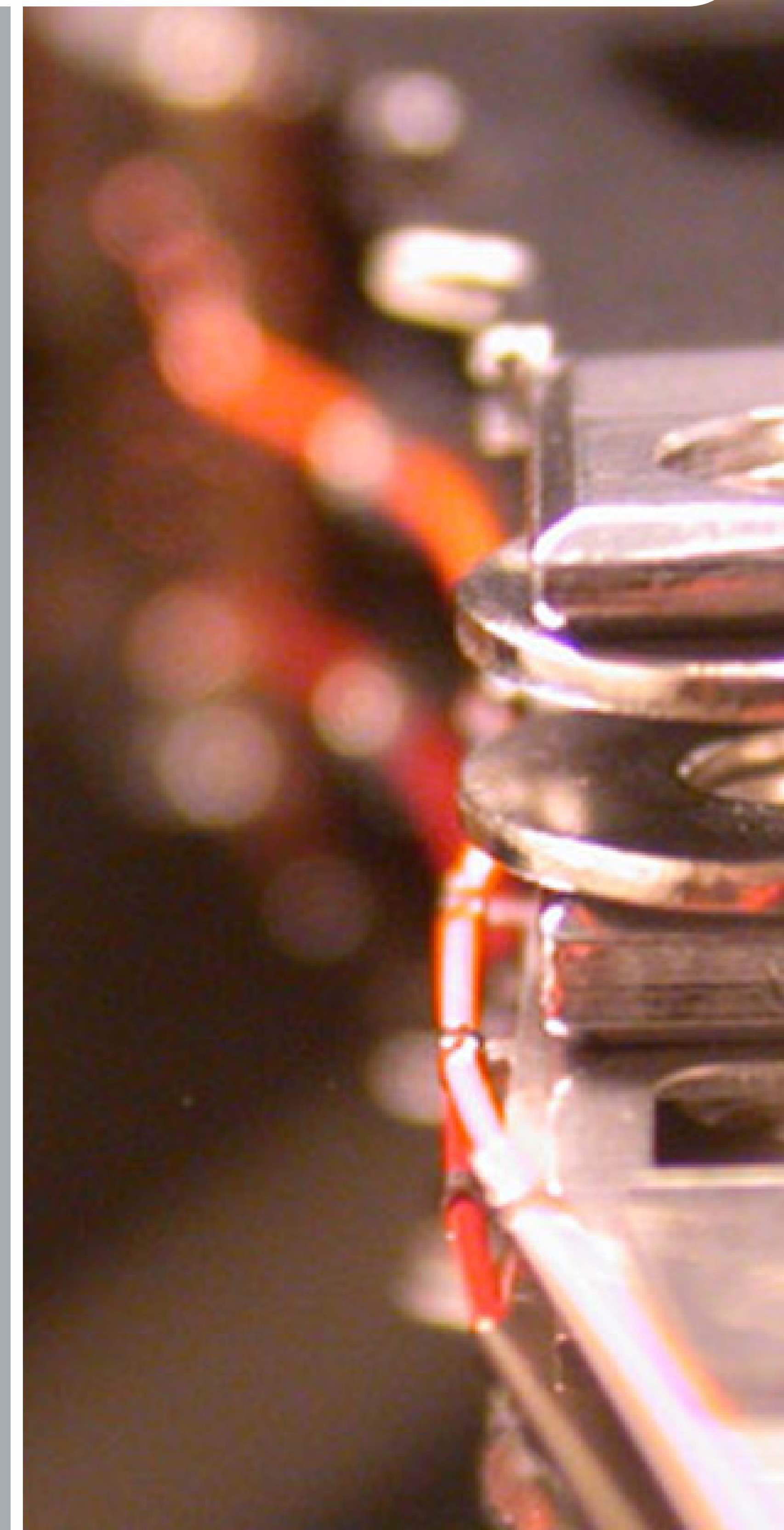
```
<!DOCTYPE html ...>
<html>
 <head>
  <title>The Little Teapot</title>
 </head>
 <body>
  <h1>The Little Teapot</h1>
    <p>There once was a little
teapot who lived all alone under
a sycamore tree.</p>
    <p>[...]</p>
 </body>
</html>
```



Tutorial: Templating (cont.)

- Some of the other included directives:
    - **call:** call one or more methods on the associated objects and insert the returned value

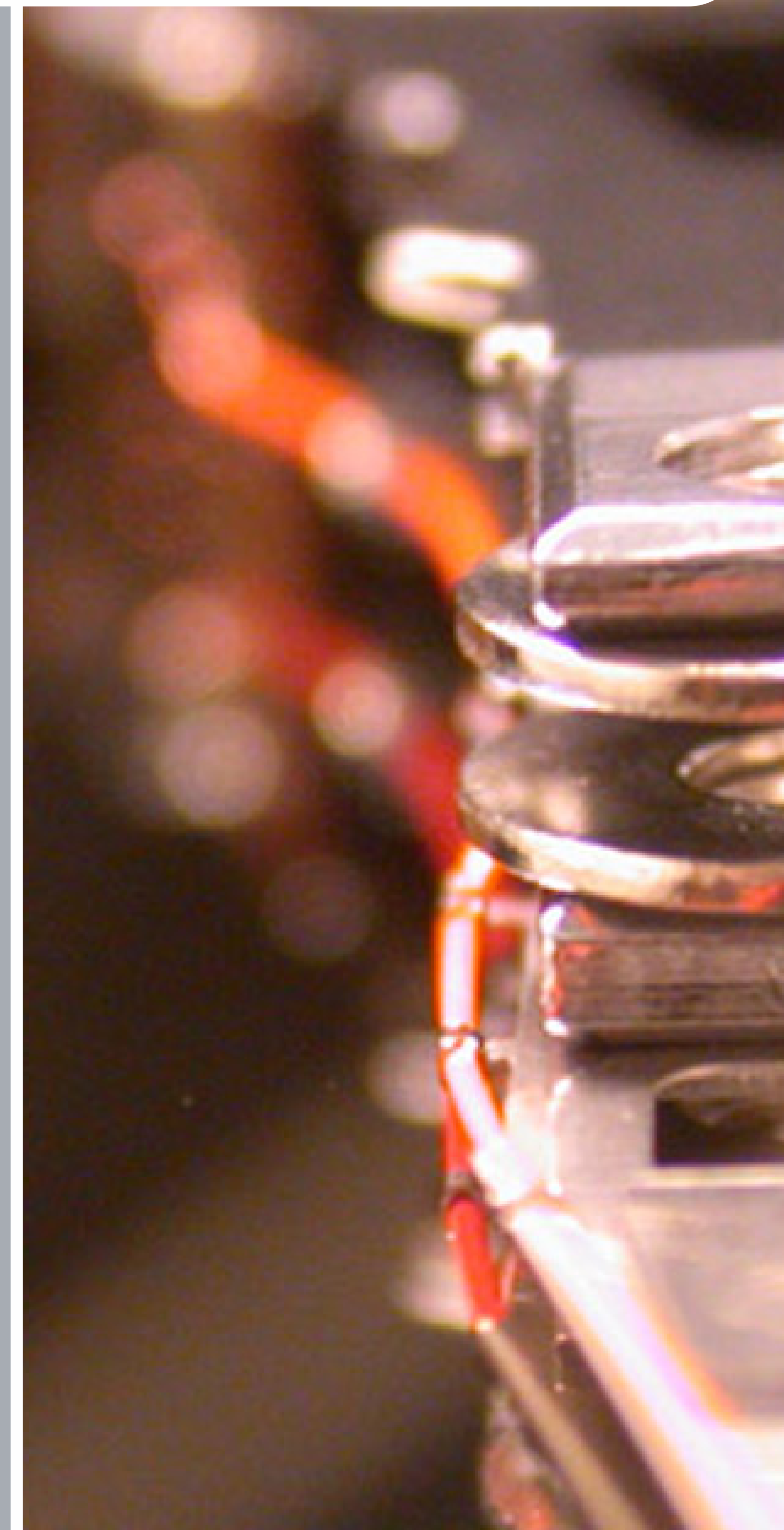
```
<?call client.name.first ?>
<?call “%0.2f” % acct.balance ?>
```



Tutorial: Templating (cont.)

- **yield:** call a method on the associated objects which takes a block and render the contents for each value it `yields`.

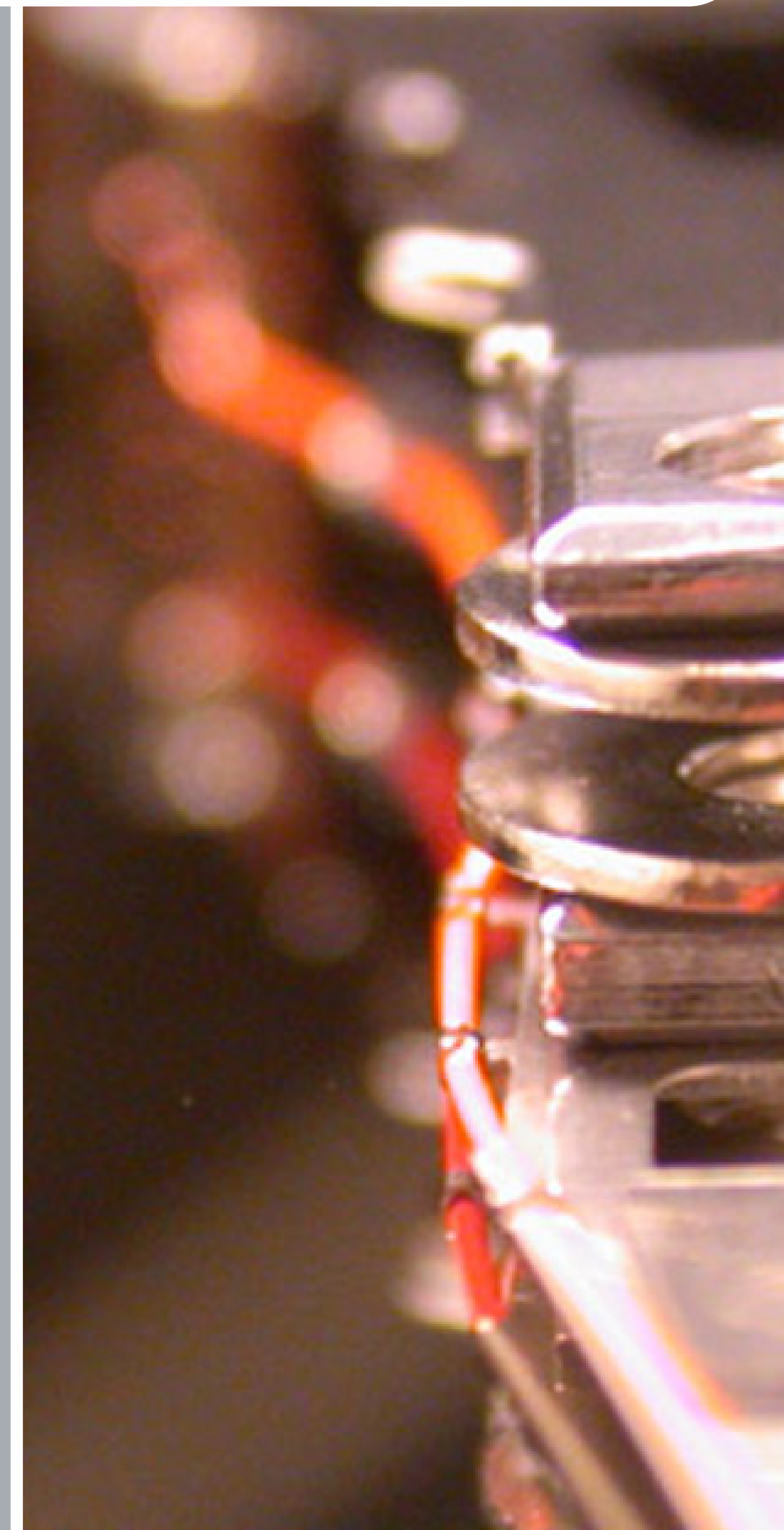
```
<?yield row from sth.fetch ?>
  Name: <?call row["name"] ?><br/>
  Address: <?call row["address"] ?>
<?end yield?>
```



Tutorial: Templating (cont.)

- **escape:** behaves like **call**, but HTML-escapes any returned stringified value:

```
<pre>
<?escape client.inspect ?>
</pre>
```



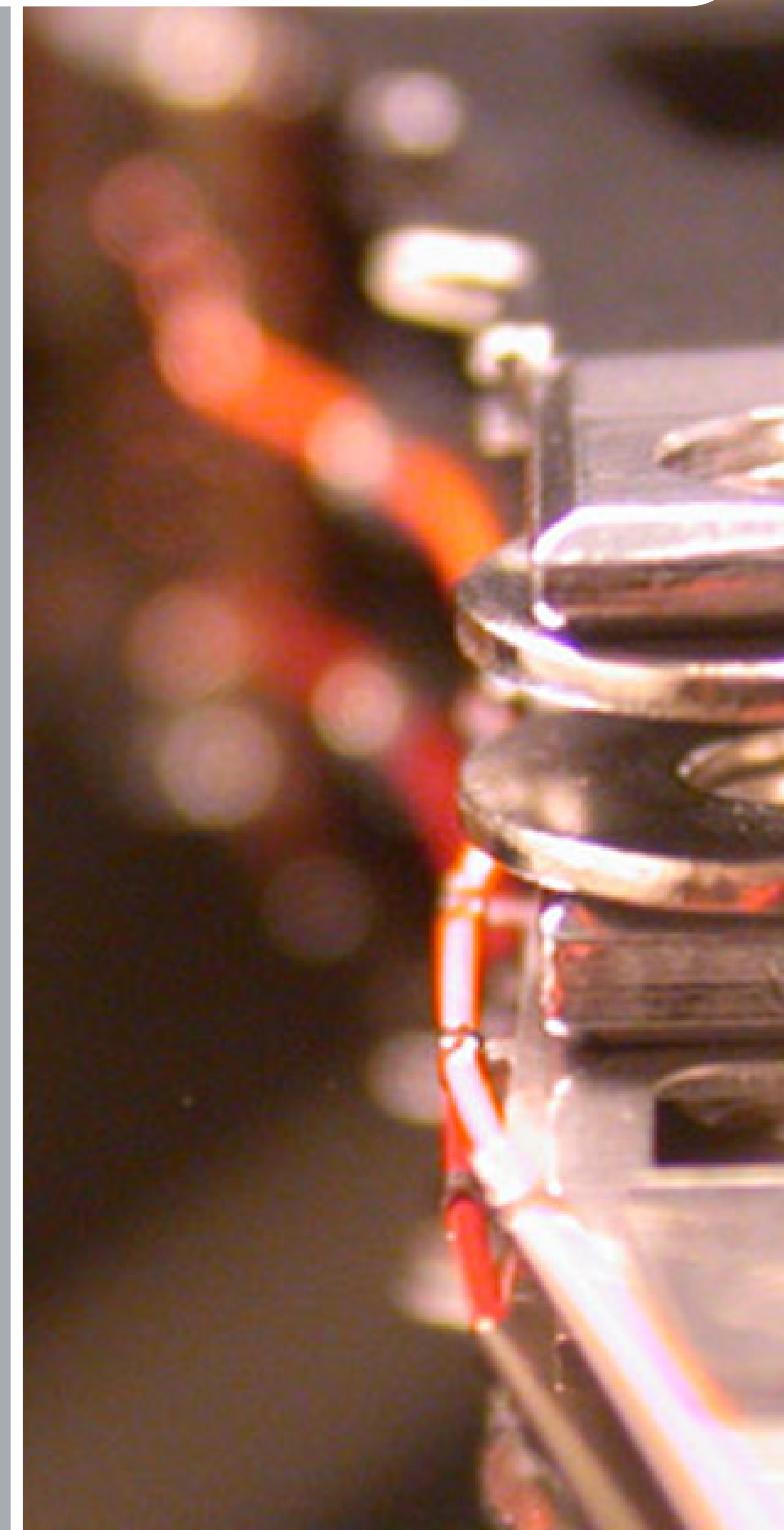
Tutorial: Templating (cont.)

- **if/elsif/else:** conditional blocks:

```
<?if client.projects.empty? ?>
<strong>No projects found.</strong>
<?end if?>
```

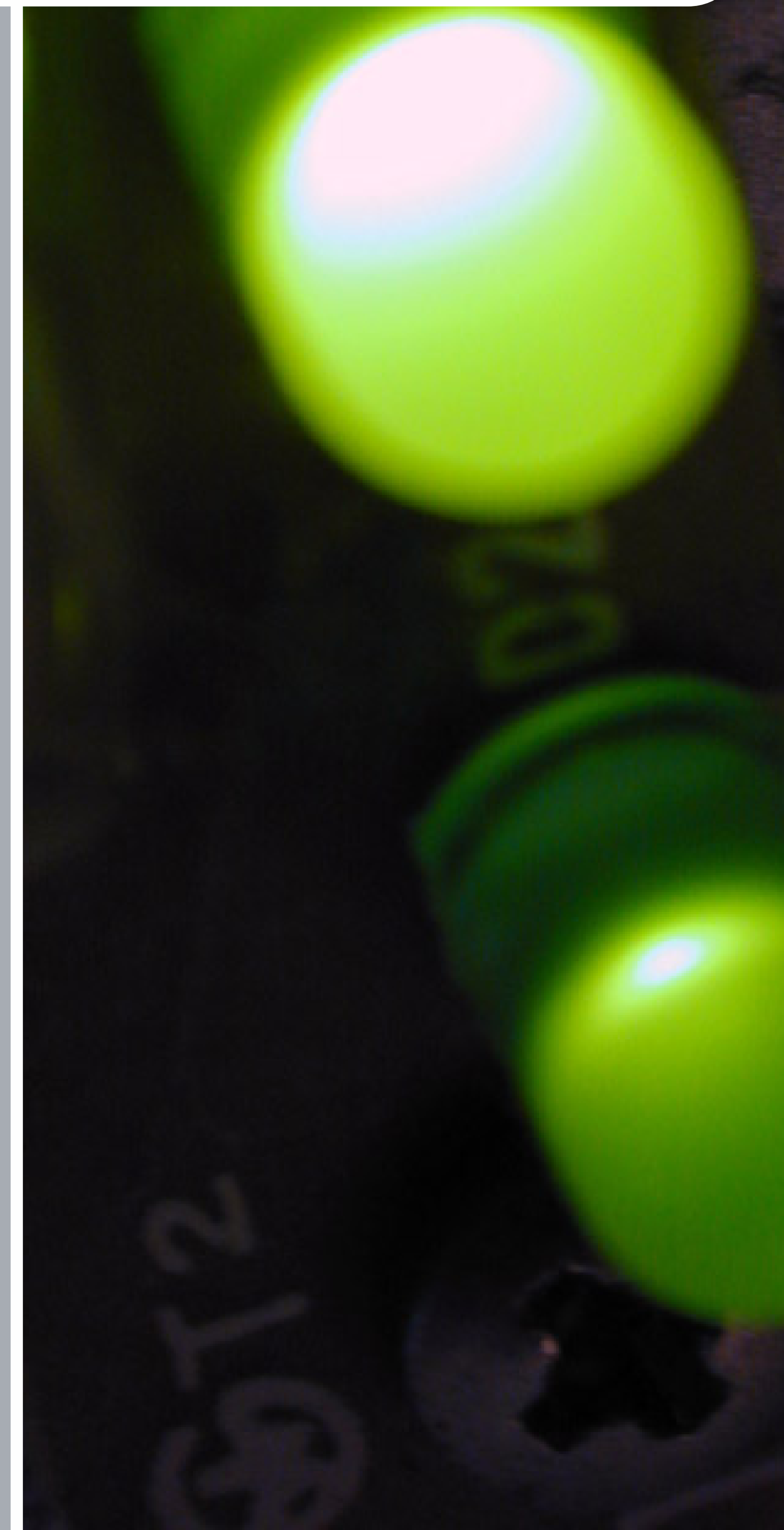
- **include:** include other templates:

```
<?include navbar.incl ?>
<?include menu.incl as topmenu ?>
```



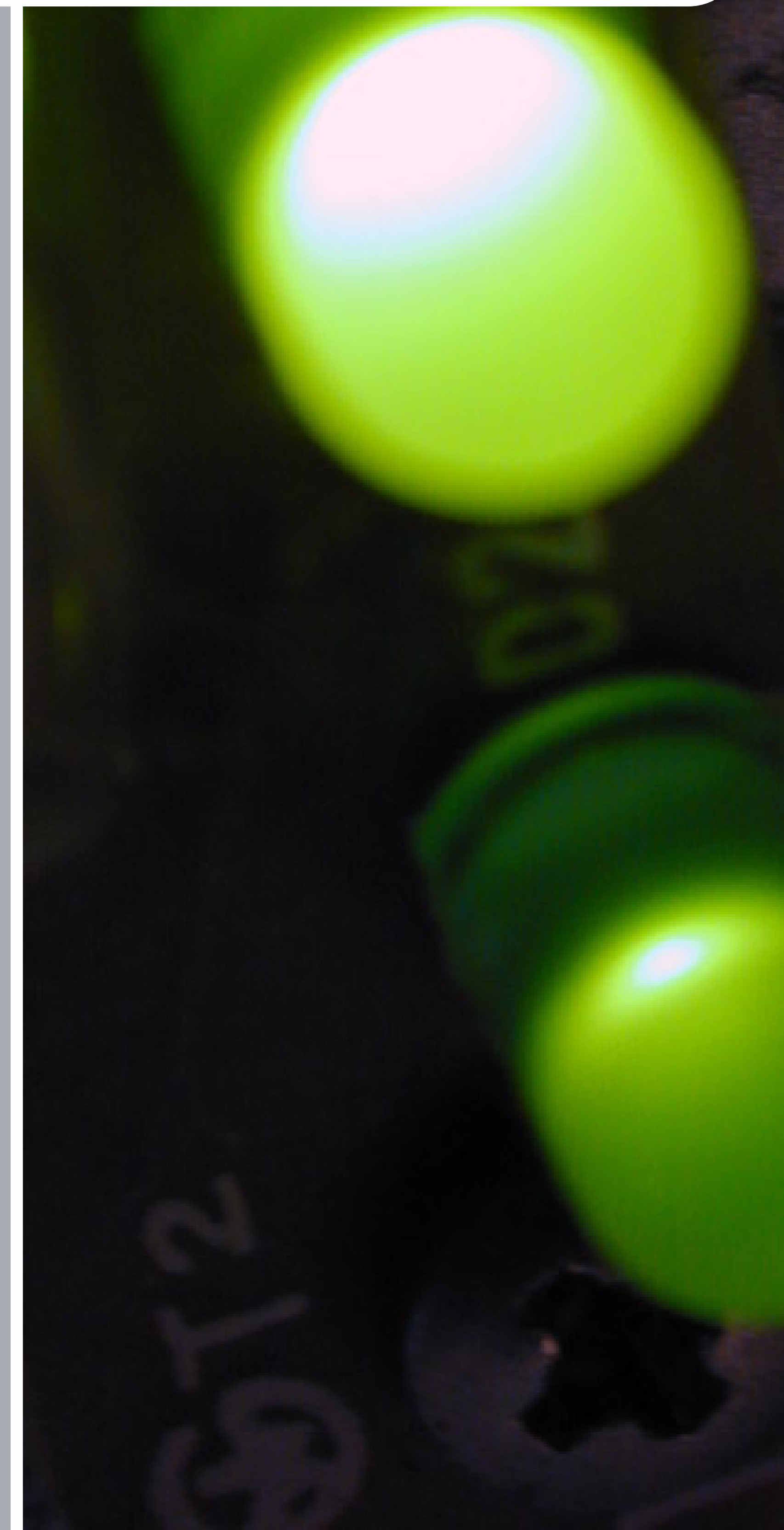
Tutorial: Sessions

- Based on Perl's Apache::Session
- Three parts:
  - Lock
  - Store
  - Id



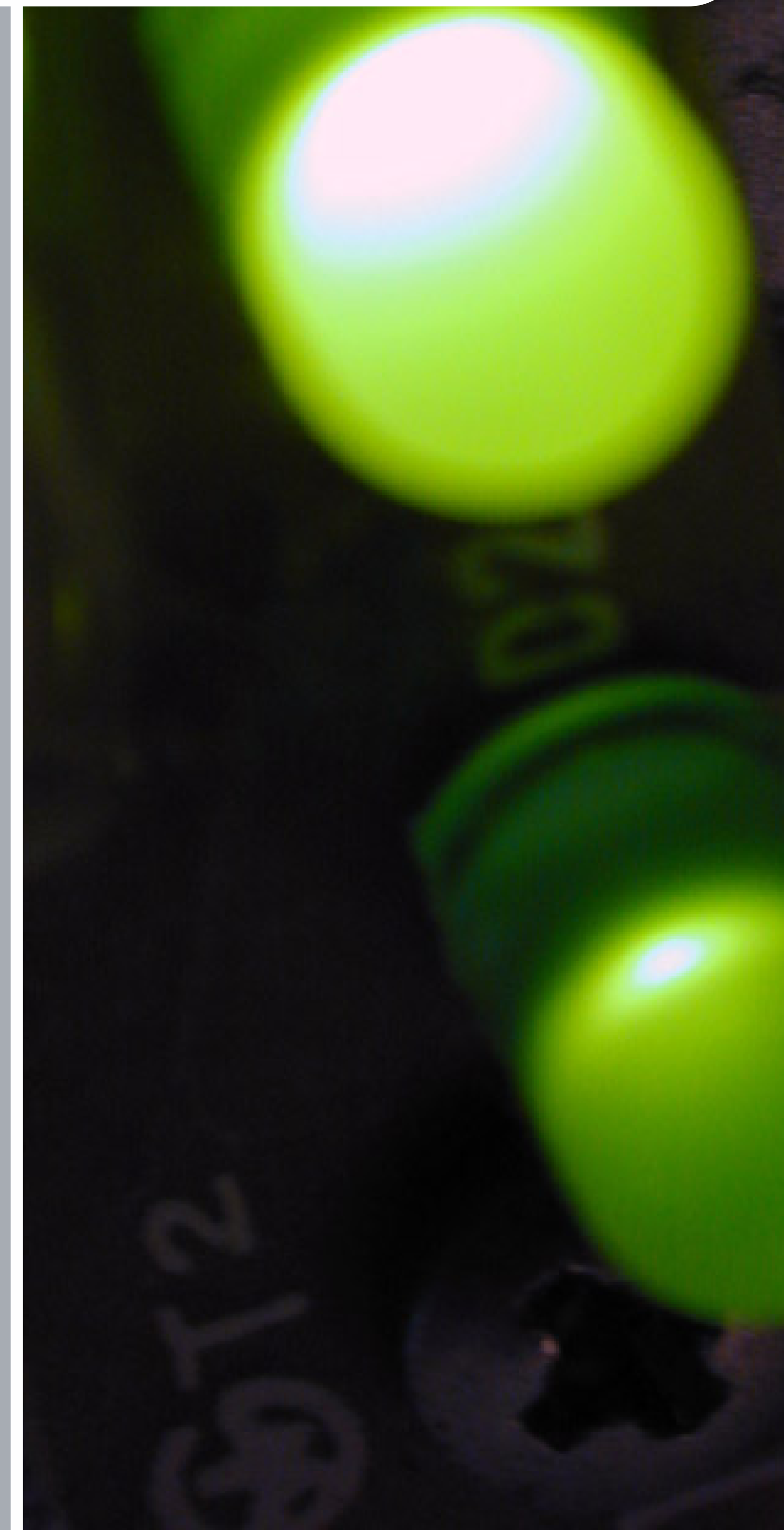
Tutorial: Sessions (cont.)

- Each part is customizable via the same plugin system that the templates use: i.e., just drop a file into a directory
- Arrow comes with a small selection of session types; more to come



Tutorial: Sessions (cont.)

- Sessions are lazily-generated
- If you don't use a session, nothing is created/loaded
- Session id is saved via a cookie currently; URL-rewriting sessions in the works



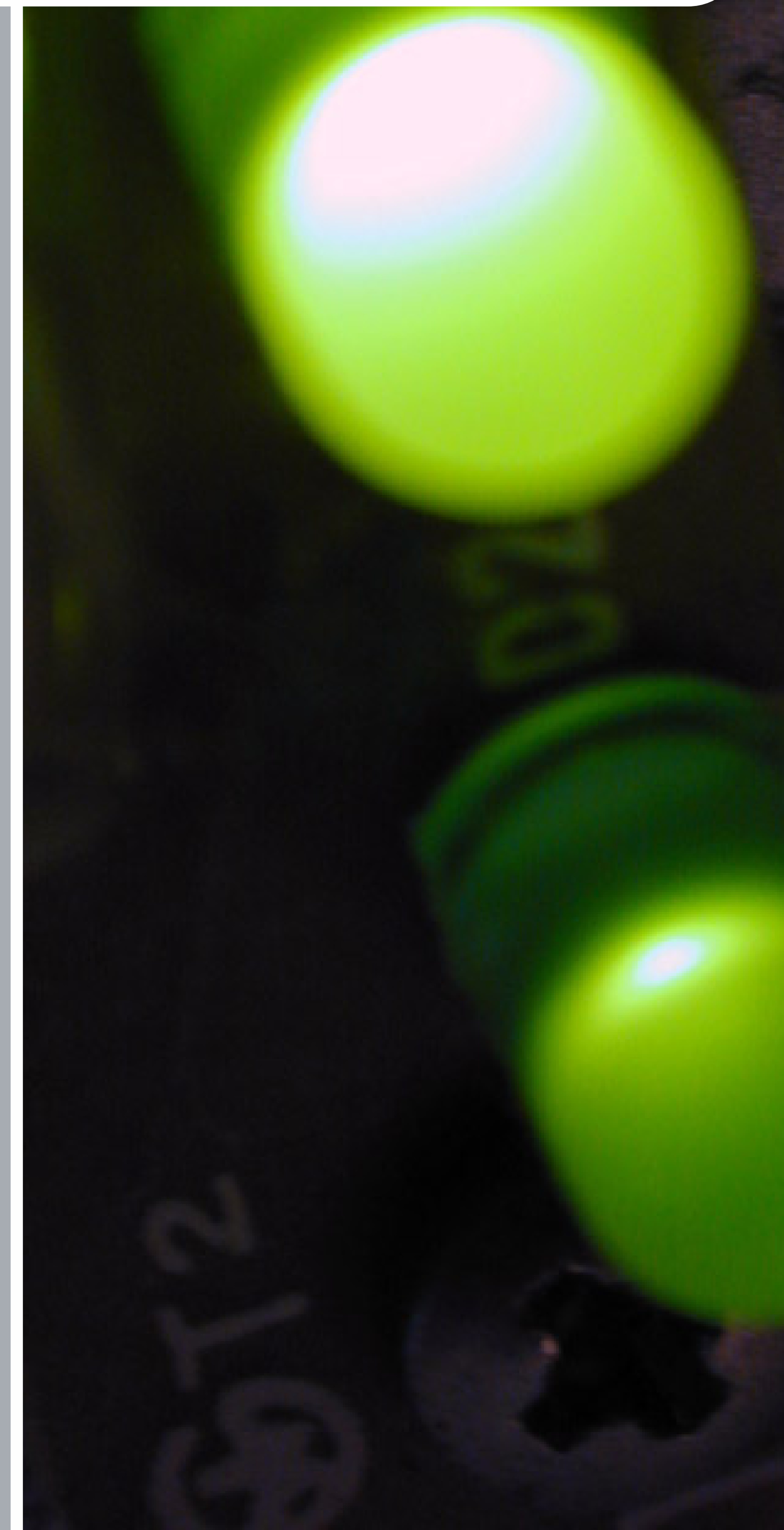
Tutorial: Sessions (cont.)

- To create/load a session, just access it via the Transaction object:

```
sessid = txn.session.id
```

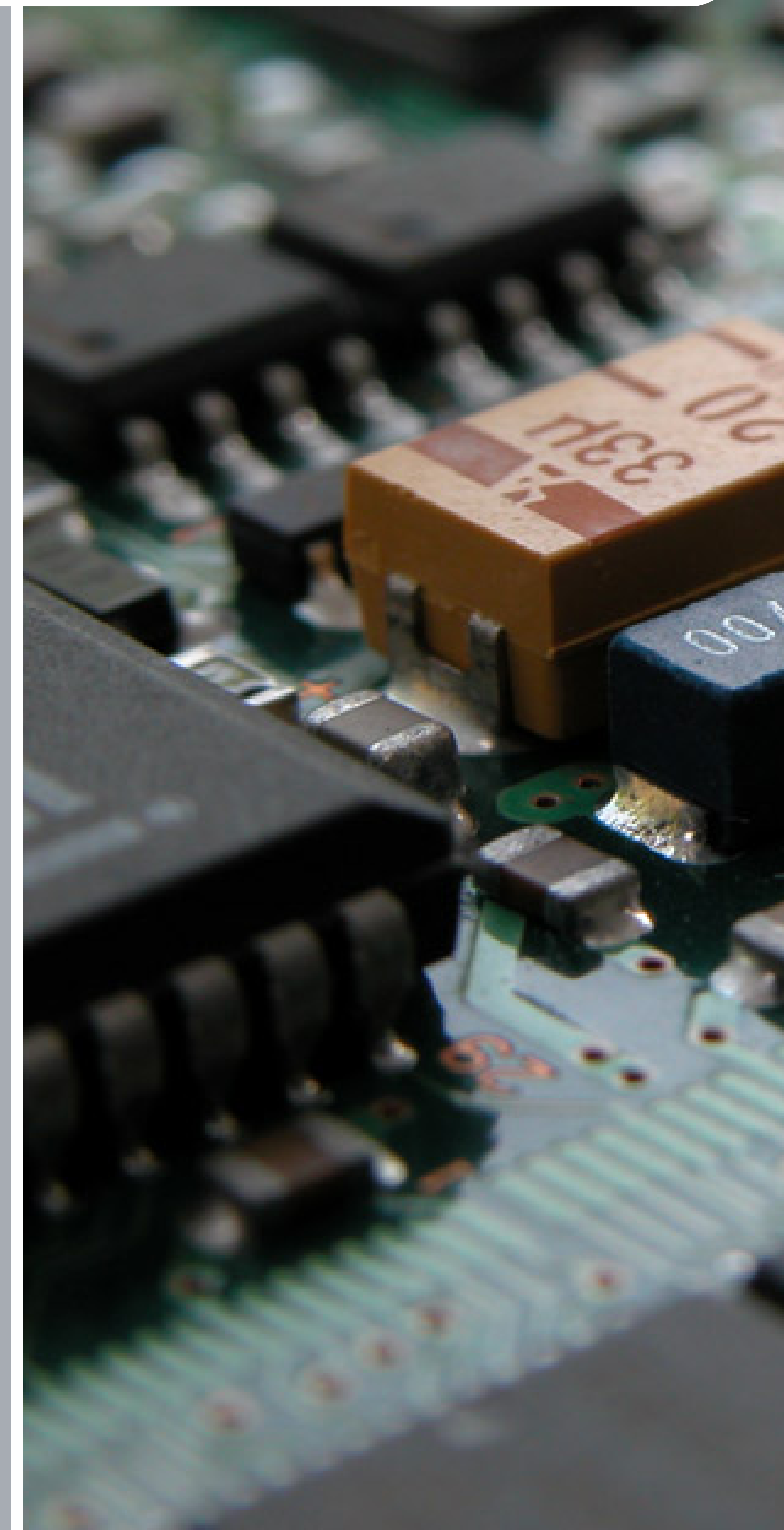
- Treat the session object like a Hash for saving values:

```
txn.session[:logged_in] = true
txn.session[:counter] += 1
```



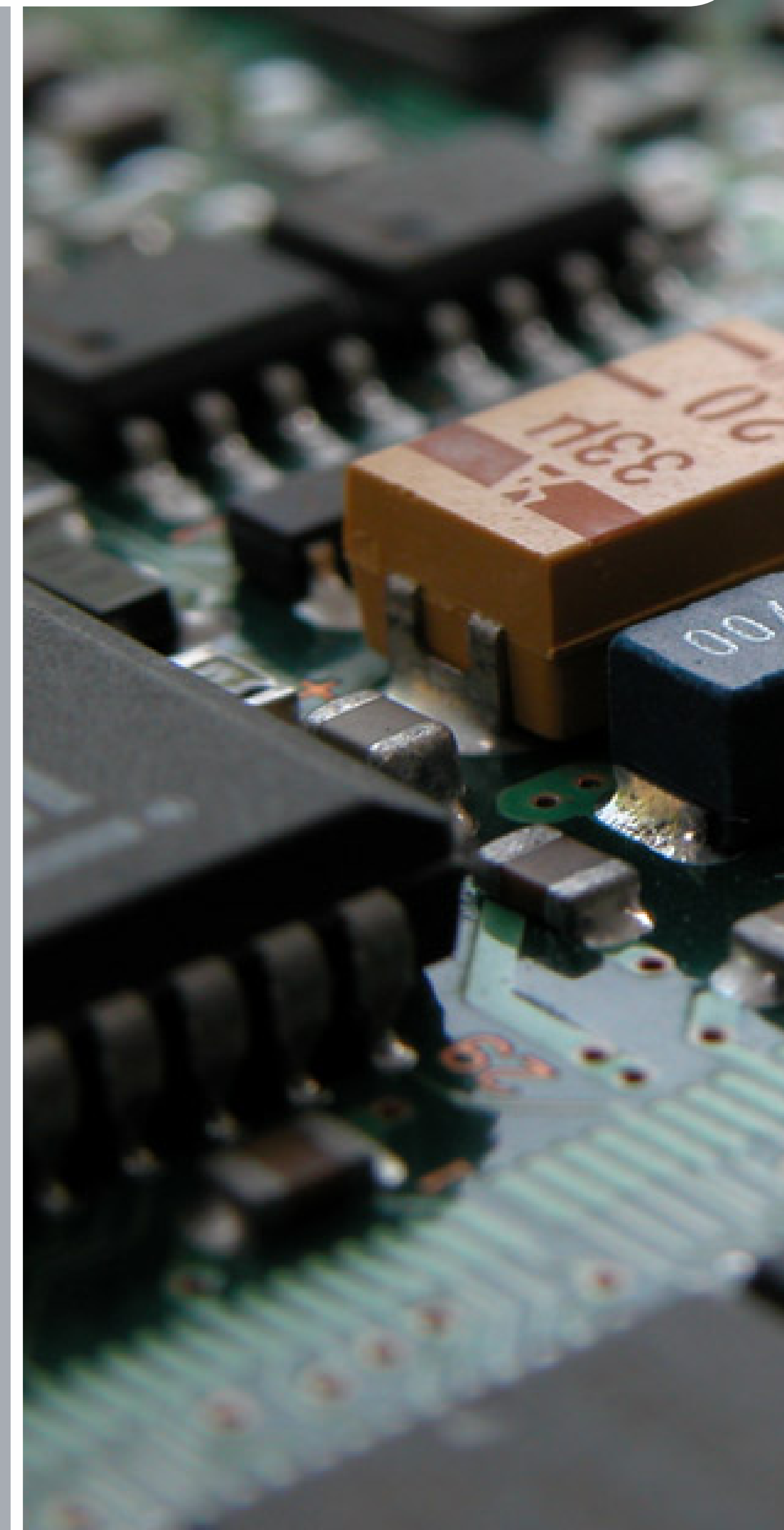
Tutorial: Argument Validation

- Untainting and validating input is a pain in the arse
- Enter FormValidator
- Integrated into Transaction



Tutorial: Argument Validation (cont.)

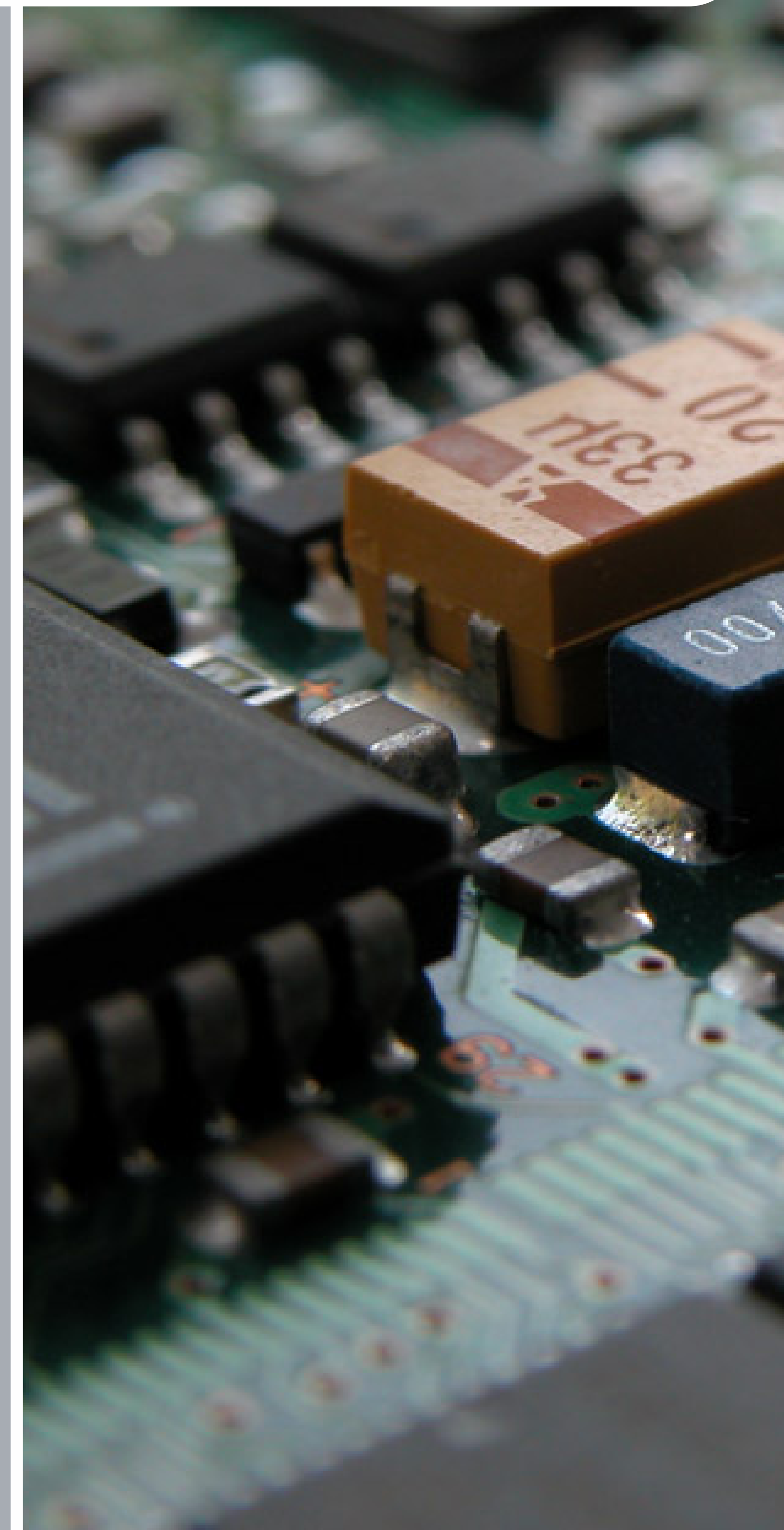
- FormValidator separates arguments into four categories: `valid`, `invalid`, `missing`, and `unknown`
- Has many more very useful features: automatic untainting, field filters, inter-field dependencies, and many other timesaving stuff.
- Worth checking out even if you don't use Arrow



Tutorial: Argument Validation (cont.)

- Arguments are configured via the applet's signature:

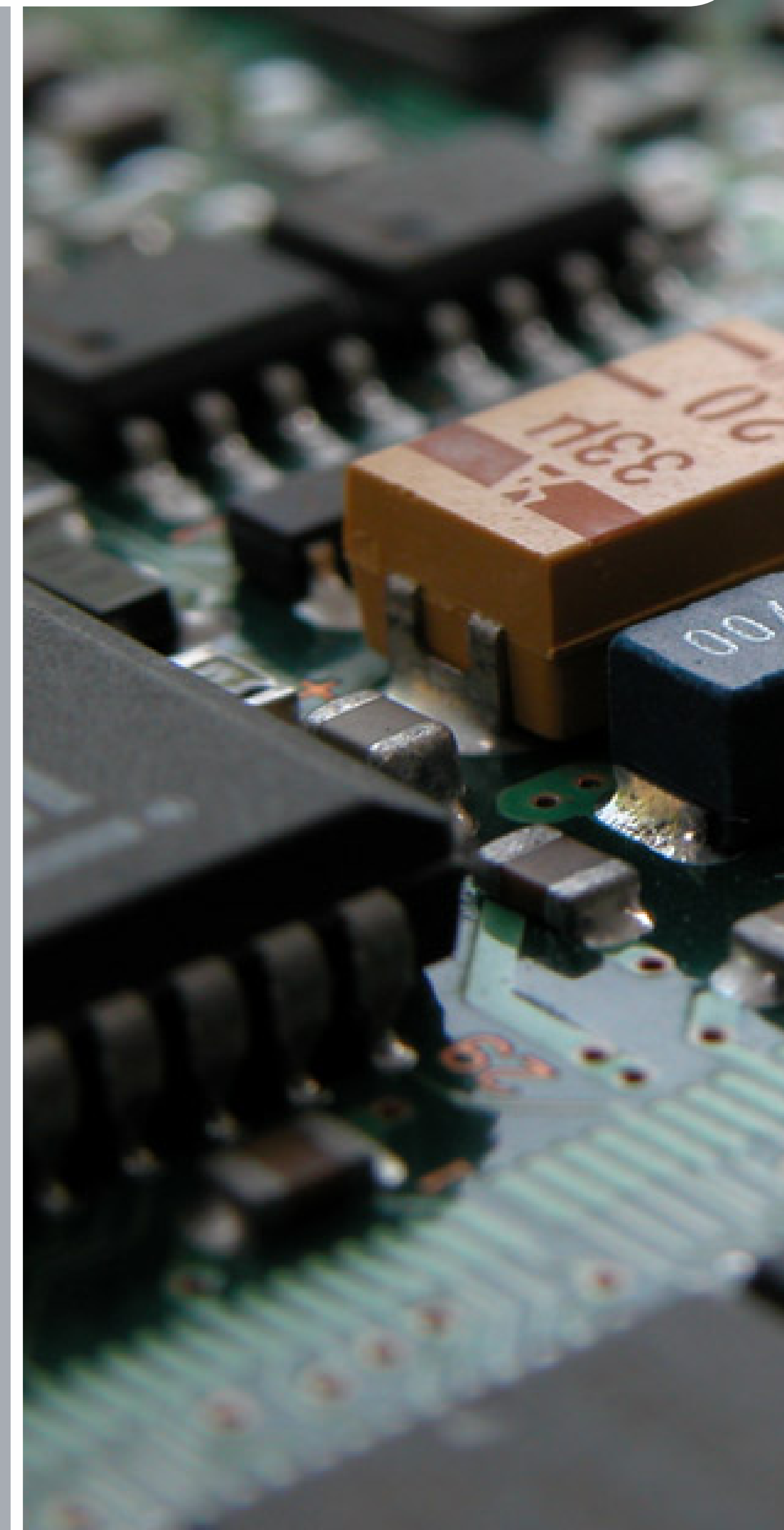
```
:vargs => {
  :display  => {
    :required    => [:name, :email],
    :optional    => [:description],
    :filters    => [:strip, :squeeze],
    :untaint_all_constraints => true,
    :constraints  => {
      :email  => :email,
      :name  => /^[\x20-\x7f]+$/,
      :description => /^[\x20-\x7f]+$/,
    },
  },
}
```



Tutorial: Argument Validation (cont.)

- Arguments can then be fetched from the Transaction's `#vargs` method:

```
#!ruby

args = txn.vargs
if args.valid.key?( “email” )
  client.email = args[”email”]
elsif args.invalid.key?( “email” )
  tmpl.error << “Invalid email”
elsif args.missing.include?( “email” )
  tmpl.missing_fields << “Email address”
end
```



Demo

- And now, a short demo.

More Info

More info, download, documentation at:

http://www.rubycrafters.com/projects/Arrow/