Using Hypermedia Services for Systems Integration

Tim Ewald
Systems Engineer

I dig REST services

- My first choice when...
  - I don't build both ends of pipe
  - I do build both ends of the pipe, but I can't deploy them en masse

Real, pragmatic benefits

- Evolution
- Routing
- Caching
- App recovery
- Vendor-free tooling

Evolution

- Self-delimiting markup lets data evolve
- Document-centric mindset is key
  - Use what you need, ignore the rest
  - Only breaks when it has to
- Hypermedia extends this, lets behavior evolve

Evolution w/ markup

```
<ns2:assets xmlns:ns2="..." xmlns="...">
  <ns2:asset>
    <ns2:id>10</ns2:id>
    <ns2:title>foo</ns2:title>
  </ns2:asset>
</ns2:assets>
```

```
<ns2:assets xmlns:ns2="..." xmlns="...">
  <ns2:asset>
    <ns2:id>10</ns2:id>
    <ns2:title>foo</ns2:title>
    <ns2:description>bar</ns2:desription>
  </ns2:asset>
</ns2:assets>
```

Evolution w/ hypermedia

```
<ns2:assets xmlns:ns2="..." xmlns="...">
  <actions/>
    <ns2:asset>
      <actions>
        <action method="get"
                name="details"
                href="/assets/10"/>
      </actions>
      <ns2:id>10</ns2:id>
      <ns2:title>foo</ns2:title>
      <ns2:description>bar</ns:desript
  </ns2:asset>
</ns2:assets>
```

```
<ns2:assets xmlns:ns2="..." xmlns="...">
  <actions>
    <action method=”post” name=”create”
            href=”/assets” body=”ns2:asset” />
  </actions>
  <ns2:asset>
    <actions>
      <action method="get"  name="details"
              href="/assets/10"/>
    </actions>
    <ns2:id>10</ns2:id>
    <ns2:title>foo</ns2:title>
    <ns2:description>bar</ns:desription>
  </ns2:asset>
</ns2:assets>
```

Consuming hypermedia

```
assets = RESTUtil.invoke :get, 'http://localhost:9292/assets'

assets.asset.each do |asset|
  puts "#{asset.get_id}, #{asset.title}"
  # get detailed description if service supports that action
  puts asset.actions.details.description
    if asset.actions.respond_to? :details
end

# create an asset if service supports that action
if assets.actions.respond_to? :create
  asset = com.schange.schemas.atria.assets.Asset.new
  asset.title = "baz"
  asset.description = "baz"
  details = assets.actions.create asset
  puts "#{details.get_id}, #{details.title}, #{details.description}"
end
```

Caching

- Caching helps scaling
  - Gives server *some* control over load
    - Decouples client load from back-end
- Output caching on server farm
- Client caching, or at least caching proxy
  - Need this for Etags to be useful

Routing

- Hypermedia controls let service control where client sends requests
  - Dynamically generated based on any criteria
- Server topology can change w/o updating clients
  - Adding bulkheads
  - Trying out new features
  - Maintenance

App recovery

- Simple pattern for resolving lost responses
  - Slightly more complex for async pattern
- Idempotent operations repeated as necessary
  - Save durably to restart after crash
- POST-exactly once semantics if necessary

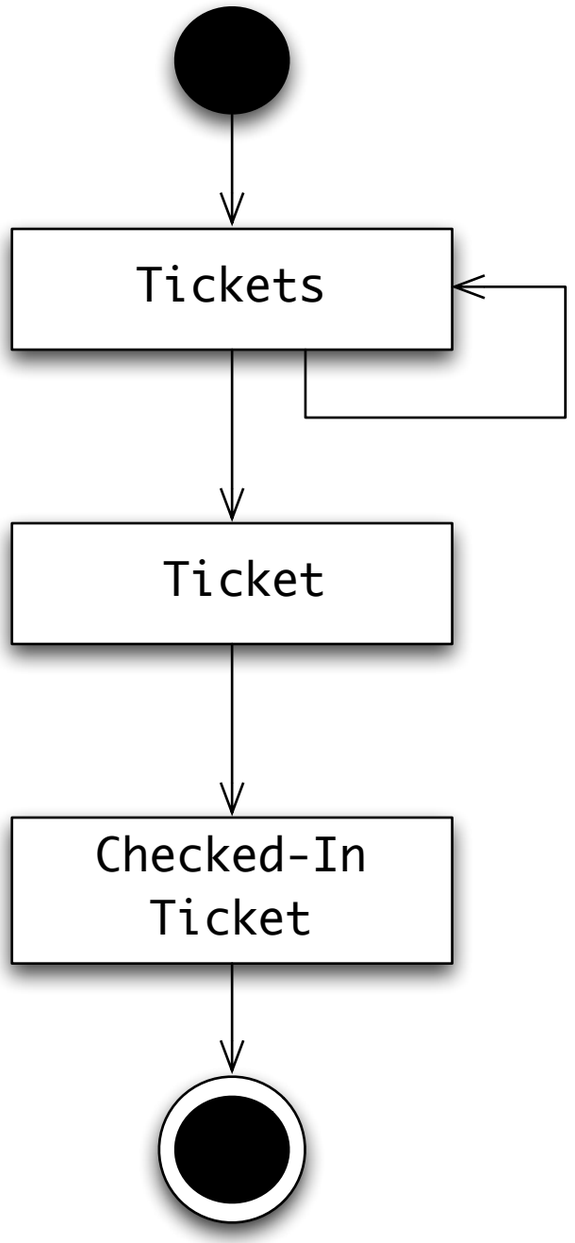
Ok, so how to build it?

- Build (at least) one yourself!
- Four steps...
  - Design protocol state machine
  - Design representations
  - Build service
  - Build client
- Keep the visual Web in mind, but understand the critical difference

Protocol State Machine

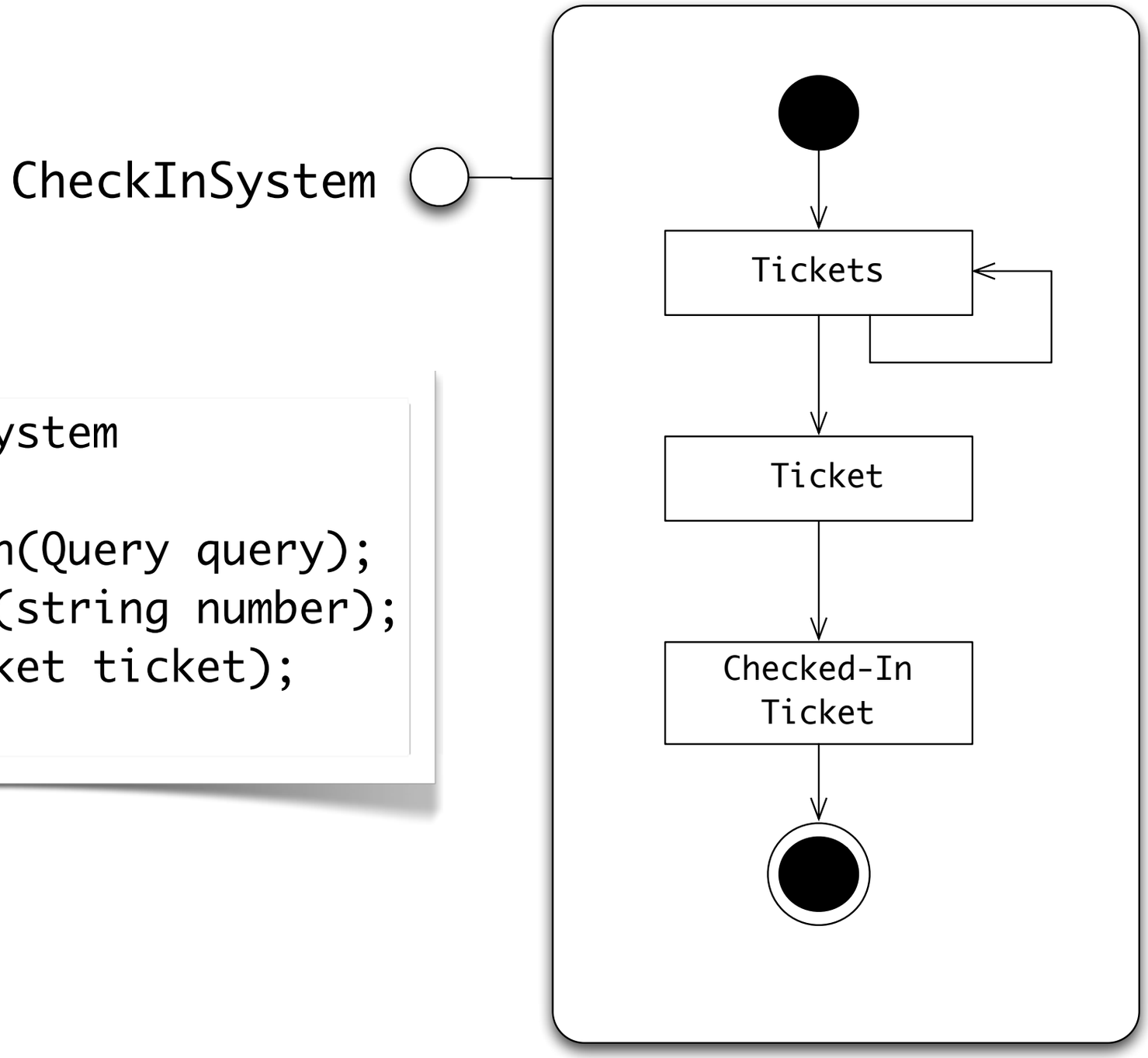
- A way to model about conversations
  - Separate from model for app state
  - Hidden in RPC behind method invocations to single endpoint
  - Exposed in REST by explicit requests to linked endpoints

Ticket PSM



RPC hides PSM

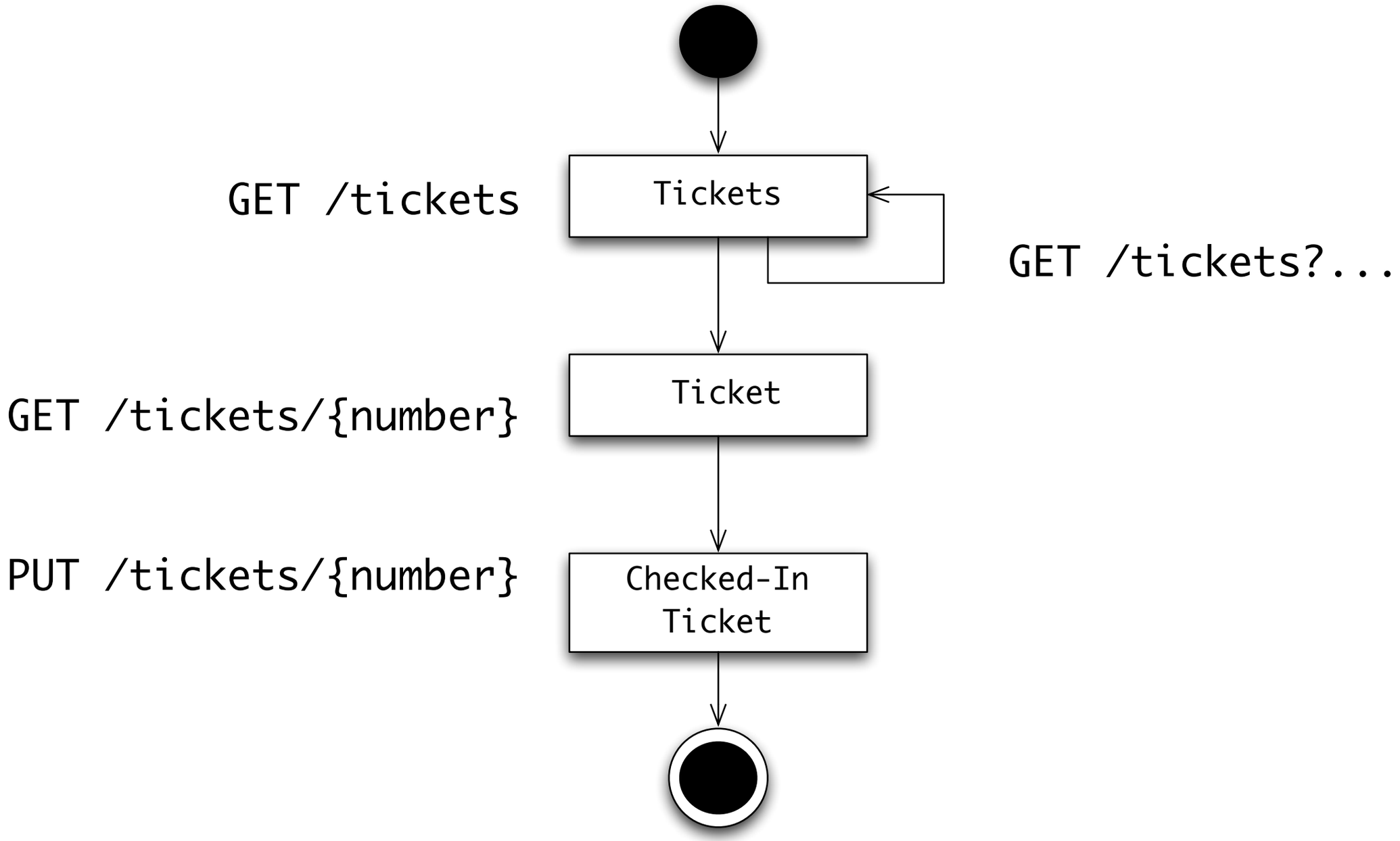
```
interface CheckInSystem
{
  TicketList Search(Query query);
  Ticket GetTicket(string number);
  bool CheckIn(Ticket ticket);
}
```



REST exposes PSM

- PSM is graph of resources
- Resource representations includes hypermedia controls forms for valid transitions
- States traversed via HTTP requests
- Current conversation state is latest {verb, URI, representation}

REST exposes PSM



Resources

- Structure reflects external projection of info model
- Use standard self-delimiting syntax
  - XML, XHTML, JSON, Clojure
- Actual structure varies by service
  - Implies out-of-band knowledge
  - MIME type necessary, but insufficient

Hypermedia controls

- In-band or out-of-band description of controls?
  - How decoupled can client/service be?
  - Is in-band documentation enough?
- Links with semantics
  - xlink, href, something custom
  - URL templates, query params?
- Forms are problematic

Building services

- Implement with handler or MVC framework
- Derive structure from PSM
- Must support relative URL generation for hypermedia controls
  - e.g., url_for
- Generate representation as view of data

Building clients

- Generic “shapeless” or “auto-shaped” proxy
- Must support URL resolution
  - Full tree navigation, pickling, zippers, etc
- Should support redirection
- Might support cookies, but only for auth
- Fantastic if it supports caching, or use proxy

Literate Services

- "But how do you know what a service does?"
- Service describes itself, just add prose and use it from the browser too
  - Exploration
  - Integration
  - Debugging
- Is this in-band enough?

Is hypermedia worth it?

- Provides significant flexibility, but does it cover required cases?
- Is overhead on wire, in storage too high?
- Is cognitive dissonance for developers too high?
- Is “loosely-coupled” the new “reusable”?