RESTful APIs

RESTful APIs

Sort of…

REST

REpresentational

State

Transfer

REST is

A software architecture style

Designed for long-lived interfaces

Complicated

Based on Dr. Roy Fielding's PhD thesis

Architectural Styles and the Design of Network-based Software Architectures

The **World Wide Web** is an example of a RESTful architecture

We focus on a subset of REST

REST is *resource-based*

not operation-based (like SOAP)

For example

- view bank transactions
- add a transaction
- update account details

as opposed to

- transfer money from A to B
- deposit money into an account
- sign up for online banking

The basic elements of REST

Resource ID
(*URI*)

VERB
(*GET, POST, ...*)

Resource
representation

The basic elements of REST

Resource ID
(*URI*)

VERB
(*GET, POST, ...*)

Resource
representation

URIs as resource IDs

/books/1244

/books/the-shining

/blog/2015/10/TheMeaningOfLife

/accounts/3761239240/transactions

URIs as resource IDs

- Should be long-lived
- Should be unique to a resource
- Should not expose internal IDs

The basic elements of REST

Resource ID
(*URI*)

VERB
(*GET, POST, ...*)

Resource
representation

The basic elements of REST

Resource ID
(*URI*)

VERB
(*GET, POST, ...*)

Resource
representation

- Single or collection resources
- Full or partial data representations
- Typically JSON, XML or HTML
  - but not limited to those

Example

```
{
  "title": "The Shining",
  "author": "Stephen King"
}
```

JSON

```
<book>
  <title>The Shining</title>
  <author>Stephen King</author>
</book>
```

XML

The basic elements of REST

Resource ID
(*URI*)

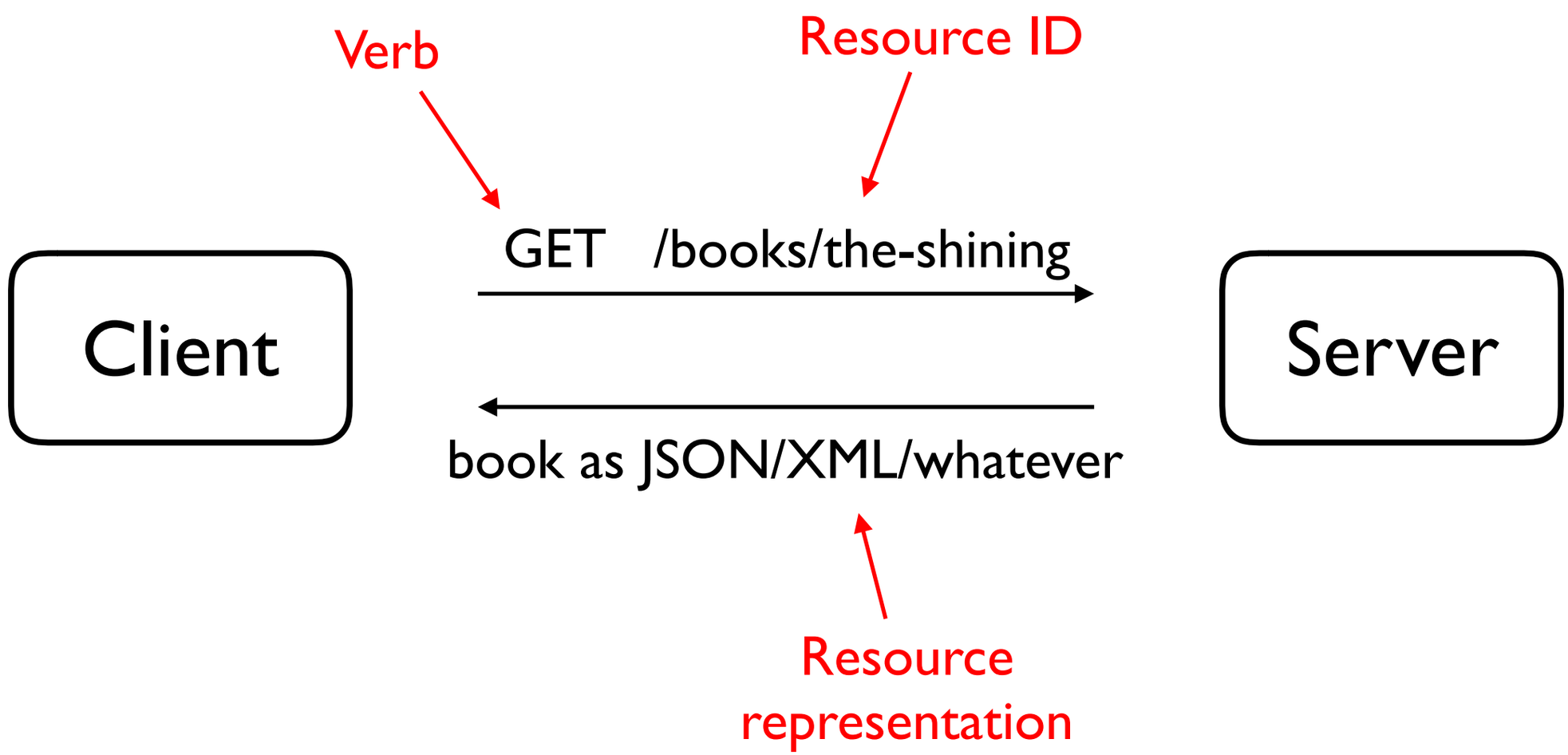
VERB
(*GET, POST, ...*)

Resource
representation

HTTP verbs

GET	*Retrieves a resource or resource collection. No side effects.*
POST	*Creates a new resource when the ID is not known in advance.*
PUT	*Creates or updates a resource when the ID is known in advance.*
DELETE	*Deletes a resource.*

Putting it together



REST in Grails

Dynamic REST “scaffolding”

```
package org.example

@grails.rest.Resource(uri="/books")
class Book {
    String title
    String author

    static constraints = {
        ...
    }
}
```

Dynamic REST "scaffolding"

URI	Verb	Action	Description
/books	GET	`index`	Retrieves all books
/books	POST	`save`	Creates a new book initialised with the given data
/books/3	GET	`show`	Retrieves the book with the given ID
/books/3	PUT	`update`	Modifies the data of the given book
/books/3	DELETE	`delete`	Deletes the given book

Customise URL mappings

Controller name

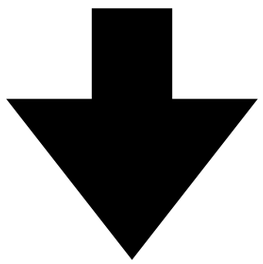
```
"/api/authors"(resources: "author") {
   "/books"(resources: "book") {
}
```

Works with
@Resource

/api/authors/121/books/42

Customise URL mappings

```
"/api/config"(resource: "appConfig")
```

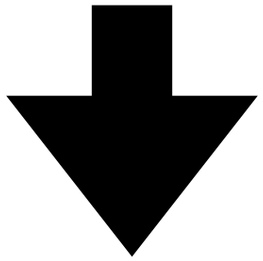


Requires a custom controller!

Verb	Action	Description
GET	`show`	Retrieves the app config
POST	`save`	Saves initial configuration (prefer PUT)
PUT	`update`	Modifies the app config
DELETE	`delete`	Deletes current app config

Customise URL mappings

```
"/api/config"(resource: "appConfig",
              excludes: ["save", "delete"])
```



Verb	Action	Description
GET	`show`	Retrieves the app config
PUT	`update`	Modifies the app config

Customise URL mappings

```
"/api/posts"(controller: "post",
             action: "allPosts",
             method: "GET")

"/api/posts"(controller: "post",
             action: "addPost",
             method: "POST")
```

Full control over mappings, but
you should favour conventions

Customise implementation

Remove `@Resource` and create the relevant controller explicitly

Customise implementation

Provides the REST scaffolding code (it's used by @Resource)

```
package org.example

import grails.rest.RestfulController

class BookController extends RestfulController {
    static responseFormats = ["json", "xml"]

    BookController() {
        super(Book)
    }
}
```

Override the standard index, show, update, etc actions here

Customise implementation

```
package org.example

import grails.converters.JSON

class BookController {
    def index() {
        render Book.list() as JSON
    }
    ...
}
```

Full control over the implementation, but you have to do all the work for error handling etc.

Content negotiation

Client *I can handle responses in either JSON or XML*

Server *OK, here's the resource as JSON*

Client *Give me the resource as XML*

Server *OK, here it is* or *Sorry, I don't know how to do XML*

Content negotiation

Client *Sends an Accept HTTP header*

Server *Renders JSON*

Client *Adds .xml suffix to URI or adds* `format=XML` *URI parameter*

Server *Renders XML* or *Returns 415 status*

Content negotiation

Comes for free with Grails if you use the `respond()` method or `response.withFormat()`

Content negotiation

```
package org.example

class BookController {
    def index() {
        respond Book.list()
    }
    ...
}
```

respond() will render the appropriate content type if it knows it. Otherwise will return the appropriate error code.

Content negotiation

```
package org.example

class BookController {
    def index() {
        response.withFormat {
            json {
                // Do something for JSON
            }
            xml {
                // Do something for XML
            }
        }
    }
    ...
}
```

Testing

Unit tests work as before

For functional tests, add *funky-spock* plugin if you don't have Geb plugin already (or equivalent)

Functional tests send HTTP requests to running app and test the responses

https://github.com/GrailsInAction/graina2/blob/master/ch12/hubbub/test/functional/com/grailsinaction/PostRestFunctionalSpec.groovy

Advanced topics

- API versioning
- Custom serialisation
- HATEOAS (links for discovering possible state transitions on resources)

Summary

- Don't go proper/full REST if you don't have to
- JSON endpoints for AngularJS don't require content negotiation
- Use appropriate HTTP status codes
- See *The Web Layer (URL Mappings)* and *Web Services* chapters of Grails user guide