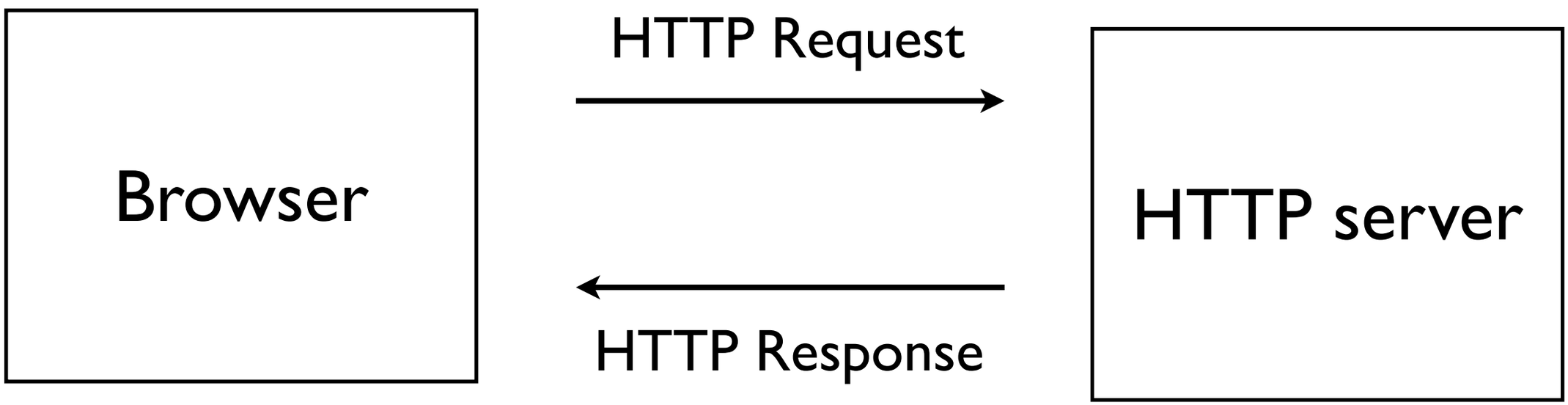
Web Development Fundamentals

It's all about HTTP



- There is *always* a request *and* a response
- The protocol is *stateless*

An HTTP request

The HTTP verb

The URI to access

A request header

```
POST /plugins/submit HTTP/1.1
Host: www.grails.org

name=shiro&version=1.3.0
```

Blank line required!

Optional request body

An HTTP response

Status code

A response header

```
HTTP/1.0 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Server: Apache/1.3.3.7
Content-Type: text/plain
Content-Length: 131
Connection: close

Hello world!
```

Blank line required!

Optional response body

Demo

The HTTP verbs

Browsers use these

- GET - fetches the content of a resource
- POST - submits new content

Common in REST

- PUT - creates or updates a resource
- DELETE - deletes a resource

- HEAD
- OPTIONS
- Others

Status codes

- 2xx - Request successful
- 3xx - Control code
- 4xx - The client has done something wrong
  - 404 Resource Not Found
  - 401 Not Authorized
- 5xx - The server did something wrong

The Abyss of Encoding

Why does my name Пётр appear as ????
or boxes?

Use UTF-8 everywhere!

HTML pages
Database
JDBC driver
Resource bundles
etc.

How does Grails let you do HTTP stuff?

Grails is...

Web Framework

MVC

Build System

HTML templating technology

GSP

Tomcat

HTTP server

GORM

H2

Database access library

In-memory & file-based database

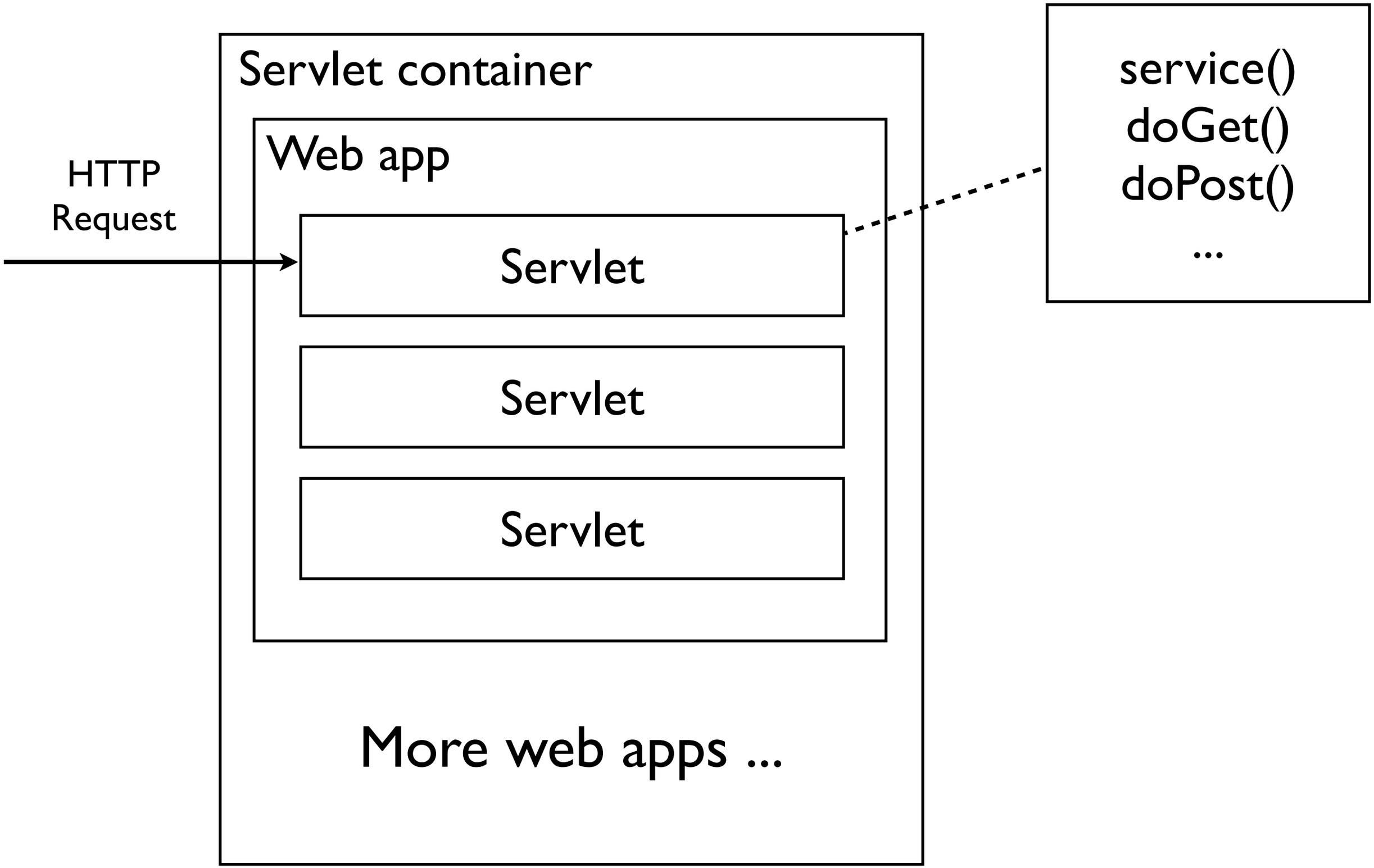
And more!

Grails is...

- A full stack framework
  - Gives you everything you need to build web apps
- Convention over Configuration
- Sensible defaults
- Based on Java standards

Demo

Servlets

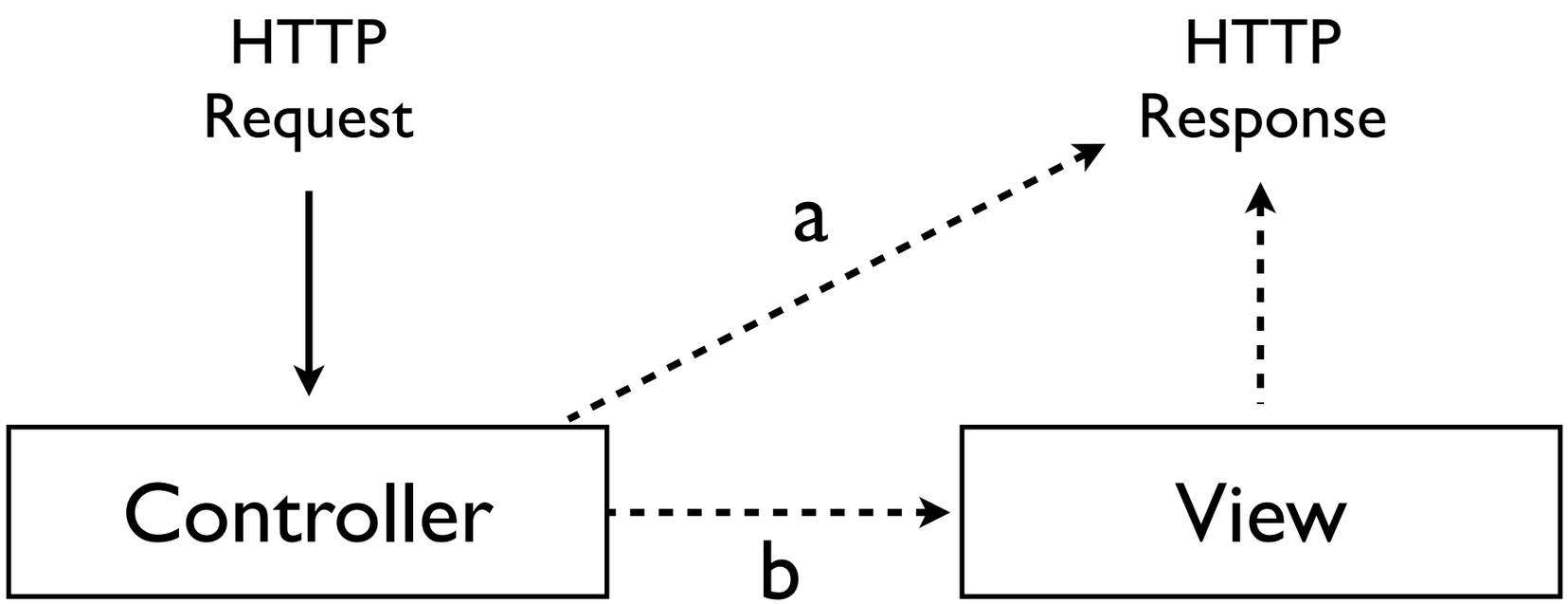


Servlets

- Part of Java Enterprise Edition (JEE)
- Servlet API is a *specification*
  - Basically defines a set of interfaces
  - A contract between servlet containers and servlet web apps
- Servlet container handles the raw HTTP
- http://docs.oracle.com/javaee/7/api/index.html

Grails

Higher level abstraction: Model-View-Controller (MVC)



a) Render content direct to response

b) Render an HTML template with a model (a set of variables and values)

Controllers

- HTTP gate keepers
- A Grails *artifact*
- Sources located in grails-app/controllers
- Class names have `Controller` suffix
- Public methods are *actions*
  - Each action handles requests to a single URL
- Can be created with `create-controller` command

Mapping requests

Optional app context

http://localhost:8080/twitter/post/show

controller

action

```
package org.example

class PostController {

    def show() {
        // Render to response
    }
}
```

A quick note on naming

Full name	org.example.PostController
Short name	PostController
Property name	postController
Logical property name	post

Important for understanding the conventions

Generating a response

Actions can use the multi-purpose `render()` method to write the response content

Simple text string - text/html

```
render "Field of Dreams"
```

```
render text: "Tron", contentType: "text/plain"
```

Simple text string with specified content type

```
render tweets as JSON
```

Converts than object to JSON and sends that in the response

Redirecting

Send client-side redirects

Redirects (302) to URI that matches this action - /post/list

```
redirect controller: "post", action: "list"
```

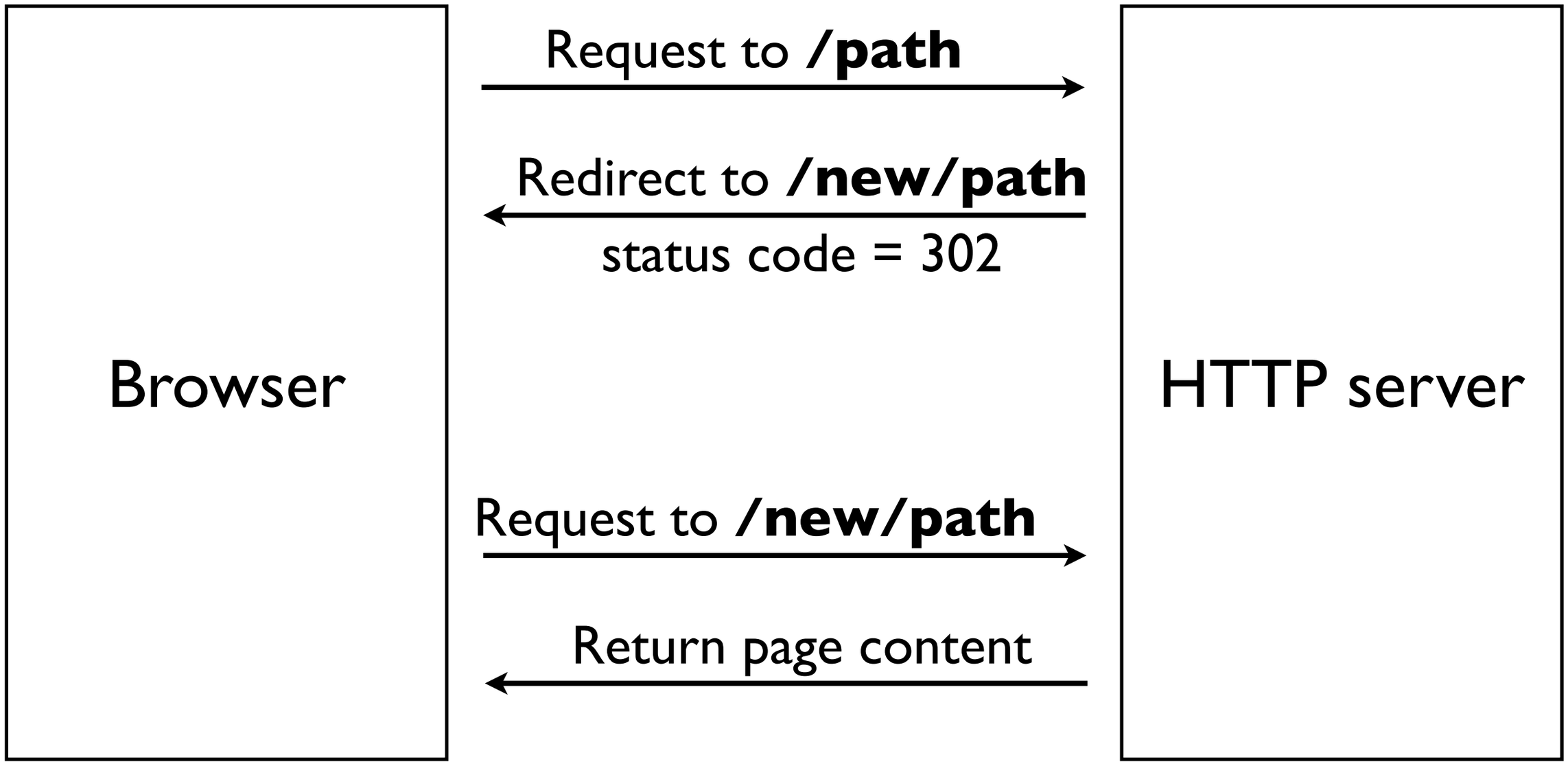
```
redirect uri: "/"
```

Redirect to URI relative to app context

```
redirect uri: "/books", permanent: true
```

Sends a 301 - Moved Permanently

What is a client-side redirect?



Implicit variables

Available to controller actions, tag libraries, filters, and views:

`params`	Map of URL parameters and form data
`request`	The HTTP request (instance of `HttpServletRequest`)
`response`	The HTTP response (instance of `HttpServletResponse`)
`session`	Per-user storage that persists between requests (covered later)
`flash`	Per-user storage that survives exactly one request and then disappears

Demo

Page rendering with views

What are views?

- Groovy Server Pages (GSP) templates
- Mix of HTML markup and Groovy expressions
- Logic encapsulated in GSP tags

Sample view

```
<html>
<head>
  <title>Books!</title>
</head>
<body>
  <h1>List of books</h1>
  <ul>
    <li>Colossus - Niall Ferguson</li>
    <li>Collapse - Jared Diamond</li>
  </ul>
</body>
</html>
```

Plain HTML!

Sample view

```
<html>
<head>
  <title>Books!</title>
</head>
<body>
  <h1>List of books</h1>
  <ul>
    <g:each in="${books}" var="b">
      <li>${b.title} - ${b.author}</li>
    </g:each>
  </ul>
</body>
</html>
```

'books' variable provided by view model

GSP tag

Groovy expression

GSPs

- `${}` delimits a Groovy expression
  - Often used for GSP tag attributes
- GSP tags are of the form `<g:tagName>`
- `<% . . . %>` block ("scriptlet") allows multiple lines of Groovy code
- `<%-- . . . --%>` delimit a GSP comment
  - Nothing inside is rendered to the page
  - No code inside is executed

Which view?

- Views are stored under grails-app/views
- Files have .gsp extension
- View selection by convention
  - Can be overridden

Which view?

```
package org.example

class PostController {

    def show() {
        return [post: ...]
    }
}
```

grails-app/views/post/show.gsp

Action returns a map as the model for the view

- View directory named after the controller
- View file named after the action

Demo

Common tags

- `<g:each>` - Useful for generating HTML lists
- `<g:if>` - Makes parts of the page visible based on a condition
- `<g:link>` - Generates a hyperlink - *<a>* - for a controller/action or uri
- `<g:formatDate>` - Converts a Date instance into a formatted string

More on tags

Tags can have a body:

```
<g:each in="${books}" var="b">
  <li>${b.title} - ${b.author}</li>
</g:each>
```

Or be empty:

```
Today's date:
<g:formatDate date="${new Date()}"
              format="dd MMM yyyy" />
```

Can be called as methods:

```
Today's date:
${g.formatDate(date: new Date(),
format: 'dd MMM yyyy')}
```

Identifying 404s

HTTP Status 404 - /library/WEB-INF/grails-app/views/home/index.jsp

type Status report

message /library/WEB-INF/grails-app/views/home/index.jsp

description The requested resource is not available.

Grails always reports JSP here

Action was found, but no view

HTTP Status 404 - /library/home/test

type Status report

message /library/home/test

description The requested resource is not available.

No action was found

Summary

- HTTP is a stateless, request/response protocol
- Controllers handle requests in Grails
  - Render content directly
  - Render a view
  - Redirect to different URL
- Views are HTML templates with Groovy expressions and GSP tags

Handling data submissions

Data submission on the web

- Data can be submitted as
  - URL query parameters
  - Request body through POST & PUT
- Browsers submit data via forms
  - POST + request body
  - application/x-www-form-urlencoded
- Other clients may submit data as JSON, XML, binary or any other data format

Creating a form

```
<html>
<head>
  <title>Create a book</title>
</head>
<body>
  <form name="createForm"
        action="/library/book/create">
    <input type="text" name="title">
    <input type="text" name="author">
    <input type="submit" value="Create">
  </form>
</body>
</html>
```

Nothing to do with controller actions

Plain HTML!

When submitted

Value of form's
action attribute

```
POST /library/book/create HTTP/1.1
Host: localhost:8080

title=Empire&author=Niall+Ferguson
```

<input> name

User-provided
value

Simplified forms in GSP

```
<html>
<head>
  <title>Create a book</title>
</head>
<body>
  <g:form name="createForm" controller="book"
          action="create">
    <g:textField name="title" />
    <g:textField name="author" />
    <g:submitButton name="create" />
  </form>
</body>
</html>
```

<g:form> generates appropriate action attribute based on controller & action

Dedicated tag for 'text' input type

Several tags for form fields

- `<g:datePicker>`
- `<g:checkBox>`
- `<g:select>` - drop-down list
- `<g:radioGroup>`
- `<g:textArea>`
- See user guide for full list of tags (including form field ones):
  - http://grails.org/doc/latest/ref/Tags/form.html

Handling form submissions

Form data

In params map

```
package org.example

class BookController {

    def create() {
        println params.title
        println params.author
        redirect uri: "/"
    }
}
```

params

- Keys and values are strings
- Contains form data *and* URL query parameters
  - e.g. /book/create?title=Colossus
- Can extract converted values:
  - params.int("yearPublished")
  - params.list("names")

Binding to types

- More structured
- Type-safe with IDE support
- Easier to understand code

Action arguments

```
package org.example

class BookController {

    def create(String title, String author) {
        ...
    }
}
```

Must match a key in the `params` map

Can be any basic type (number, character or string)

- Good for small numbers of parameters (approx. < 4)
- Can't use `Date`, `List` or any other complex type
- If `params` value does not match the argument type:
  - Argument value defaults to `null`
  - Controller's `errors` property populated with the error

Demo

Command objects

- Bind request data to a class
- Class has typed properties
- It's dependency-injected
- Has constraints
  - values can be validated
  - each instance has its own `errors` property

Example

```
package org.example

@grails.validation.Validateable
class Profile {
    String fullName
    Date   dateOfBirth
    String emailAddress
    Double height

    static constraints = {
        fullName blank: false, nullable: false
        dateOfBirth max: new Date()
    }
}
```

Must match a key in the *params* map for binding to happen

Example

```
package org.example

class AuthorController {
    def update(Profile profile) {
        if (profile.hasErrors()) { ... }
        ...
    }
}
```

Method added by Grails to `Validateable` objects (+ `errors` property)

Grails does the following:

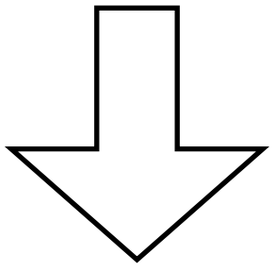
- Instantiates the `Profile` class
- Binds `params` to this object
- Calls `validate()` on the profile
- Calls the `update` action, passing in the profile as its argument

Validation errors

- In the `errors` property of the command object
- Of type `org.springframework.validation.Errors`
- Binding errors
  - Mis-match between parameter value and bound property type
- Validation errors
  - Constraint violations

Rendering errors

```
<g:renderErrors bean="${profile}" />
```



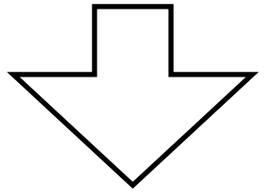
```
<ul>
  <li>Property [...] of class [...] with value
[...] exceeds maximum value [...]</li>
</ul>
```

Customising the error text

- Find the constraint's error code in the user guide
  - http://grails.org/doc/latest/ref/Constraints/Usage.html
- Add the error code to appropriate resource bundle
  - e.g. grails-app/i18n/messages.properties

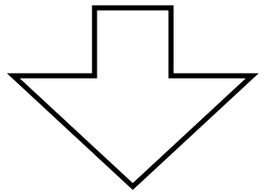
Example

Given the class `Profile`
and the property `dateOfBirth`
whose value violates the `max` constraint



Error code for `max` constraint:

`className.propertyName.max.exceeded`



Add to messages.properties:

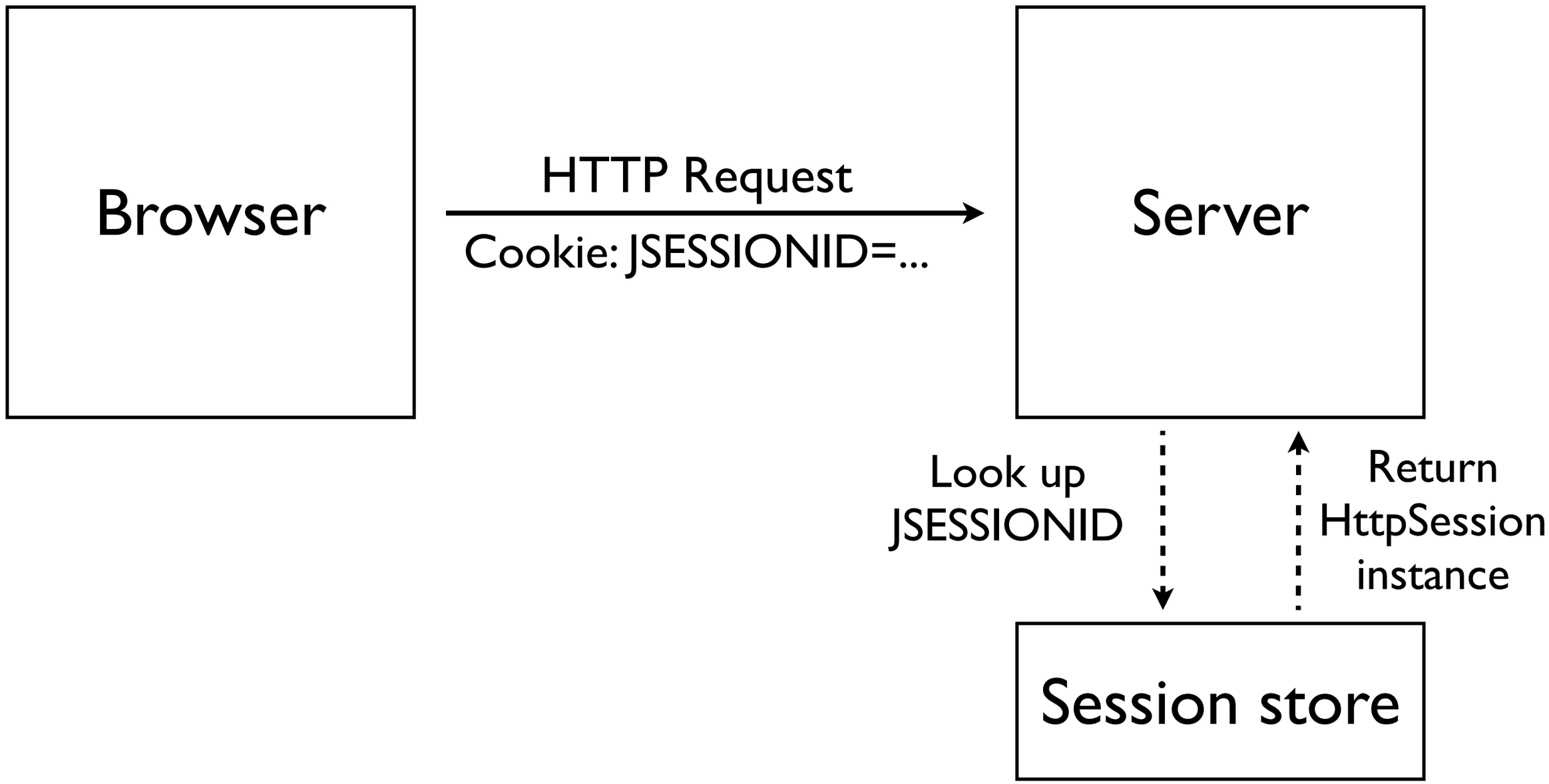
```
profile.dateOfBirth.max.exceeded=Date of birth
can''t be after today
```

Demo

HTTP Session

- HTTP is stateless
- Servlet specification adds state via a *session*
  - Data survives between requests
  - Each client gets its own session
- Treat it as a map of variable names and values
- Useful place to store submitted data for exercises

HTTP Session



Dangers of sessions

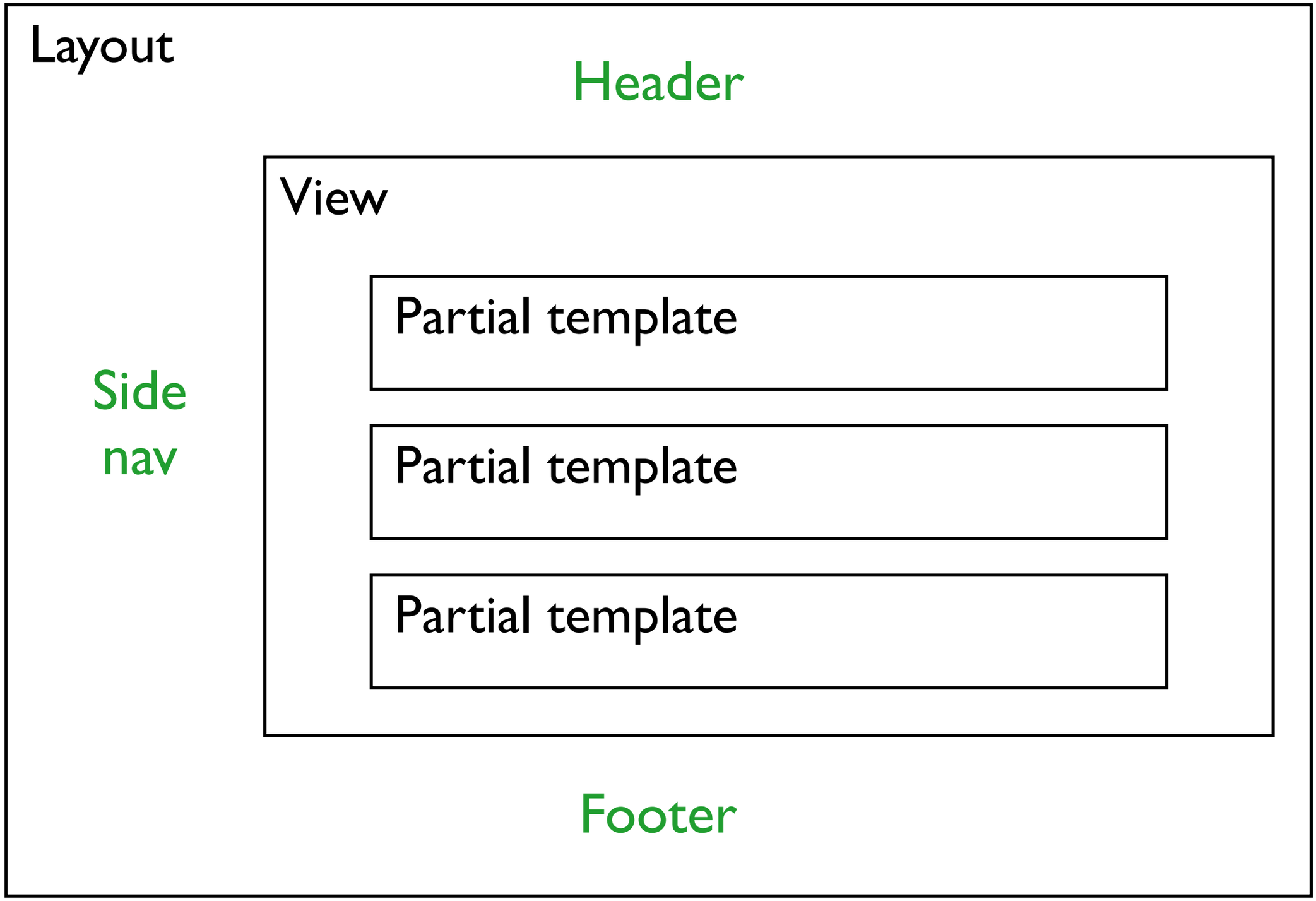
- Sessions can be hijacked if the cookie is known or guessed
- Hurts scalability once created
  - Imagine millions of users, each with a session
- Beware what you store
  - Small(ish) objects can be large(ish) in their millions
- Minimise what you store in terms of size

Summary

- Forms are submitted as HTTP POST
  - data available from `params` object
  - can be bound to typed action arguments
  - or to command objects
- Command objects have constraints
  - property values are validated
  - errors stored in the `errors` property
- Sessions add conversation state to HTTP requests

GSP features

GSP Hierarchy



Layouts

What are layouts?

- Decorator pattern for views
  - Apply a consistent look & feel to pages
- GSP templates
  - can use GSP tags, expressions, etc.
- Go in grails-app/views/layouts
- Applied *after* the view is rendered
- Has no dedicated model
  - But has access to the view's model

Typical layout GSP

```
<html>
<head>
  <title><g:layoutTitle /></title>
  <g:external dir="css" file="main.css" />
  <g:layoutHead />
</head>
<body>
  <g:layoutBody />
</body>
</html>
```

Places the view's title here

View's <head> content goes here

View's <body> content goes here

- Often links to CSS/JS files
- Layout tags are specific to layouts!

Which layout for a view?

In order of precedence:

- `<meta>` tag in the view
- By convention on action name
- By convention on controller name
- application.gsp
- None

<meta> layout

If view has layout <meta> tag:

```
<head>
  <meta name="layout" content="main" />
  ...
</head>
```

Name of the layout

This layout is applied:

grails-app/views/layouts/main.gsp

Action convention

```
package org.example

class BookController {

    def show() {
        ...
    }
}
```

Name of the action

If it exists, this layout is applied:

grails-app/views/layouts/book/show.gsp

Logical property
name of controller

Controller convention

```
package org.example

class BookController {

    def show() {
        ...
    }
}
```

Name of the controller

If it exists, this layout is applied:

grails-app/views/layouts/book.gsp

Demo

Partial templates

What are partial templates?

- GSP fragments - not entire views
- Can use tags, expressions, etc.
- Reusable in multiple views
- Useful for consistent display of a particular bit of data
  - Think cookie-cutter!
- GSP files prefixed with '_'

The source files

If a view associated with BookController has:

```
<g:render template="condensed" bean="${book}"/>
```

Name of the partial template

This partial template is used:

grails-app/views/book/_condensed.gsp

Controller name

The path is relative to the 'current' controller

The source files

If a view associated with BookController has:

```
<g:render template="/shared/user" bean="${u}"/>
```

Name + absolute path of the partial template

This partial template is used:

grails-app/views/shared/_user.gsp

The path is relative to the 'views' directory

Example

grails-app/views/book/_condensed.gsp

```
<span class="short">
  ${b.title} ${b.yearPublished}
  (${b.author})
</span>
```

Used like this:

```
<g:render template="condensed" bean="${book}"
          var="b" />
```

Variations of <g:render>

Renders template for a single object

```
<g:render template="..." bean="${book}"
          var="b" />
```

Renders template for a collection of objects

```
<g:render template="..." collection="${book}"
          var="b" />
```

Renders a template that has multiple, independent variables

```
<g:render template="..." model="[b: book]" />
```

Special case

Say you want to use GSP to render HTML emails

In a controller action:

```
def register() {
    ...
    if (successful) {
        sendMail {
            to userEmail
            subject "Successfully registered"
            body g.render(template: "registerEmail",
                          model: [u: user])
        }
    }
}
```

Call tag as if it were a method - returns a string

Demo

Custom Tags

What are tags?

- Encapsulated logic for views
- Reusable (like partial templates)
- Implemented in TagLib *artifact*
  - Class name must have TagLib suffix
  - Source file must be in grails-app/taglib
- Preferred over scriptlets (<% . . . %>)

Example - empty tag

```
package org.example

class LibraryTagLib {

    def shortDate = { attrs ->
        def formatter = new SimpleDateFormat(
                "dd MMM yyyy")
        out << formatter.format(attrs.date)
    }
}
```

Name of tag - a closure property, not a method!

A map containing the attribute values

Write content to the page

How the date attribute maps between GSP and tag impl.

Used in a GSP like this:

```
<g:shortDate date="${profile.dateOfBirth}" />
```

Example - content tag

```
package org.example

class LibraryTagLib {

    def emphasise = { attrs, body ->
        out << "<em class='mega'>"
        out << body() << "</em>"
    }
}
```

A closure

Render the tag content

Used in a GSP like this:

Handled automatically by the body() call

```
<g:emphasise>
  Sale on <g:shortDate date="${saleDate}" />!
</g:emphasise>
```

Custom namespace

```
package org.example

class LibraryTagLib {
    static namespace = "lib"

    def emphasise = { attrs, body ->
        ...
    }
}
```

This property sets the tag namespace

Used in a GSP like this:

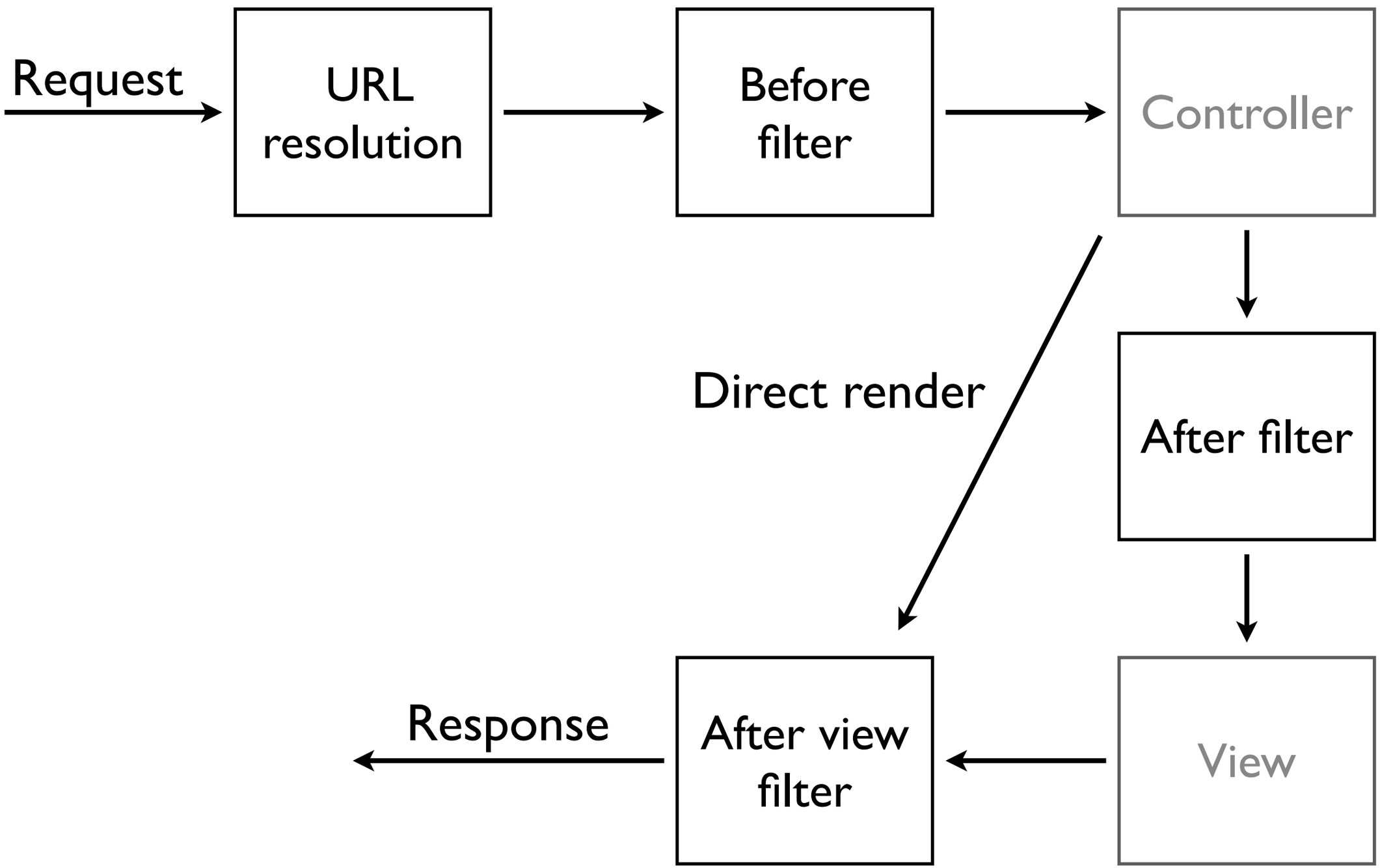
```
<lib:emphasise>
  Sale on <g:shortDate date="${saleDate}" />!
</lib:emphasise>
```

Summary

- Layouts are for markup that applies around multiple views
- Partial templates are for re-usable markup
  - multiple times in a single view, or
  - across multiple views
- Tags are for custom view logic
- Some overlap between tags and partial templates
  - The more logic, the more appropriate to use tags

Advanced request processing

GSP Hierarchy



URL Resolution

URL Mappings

Which action should handle which URL?

```
class UrlMappings {

    static mappings = {
        "/$controller/$action?/$id?(.${format})?" {
            ...
        }

        "/"(view:"/index")
        "500"(view:"/error")
    }
}
```

Convention based mapping - removing this will result in 404s everywhere!

grails-app/conf/UrlMappings.groovy

Basic usage

```
static mapping = {
    "/books"(controller: "book" , action: "list")
}
```

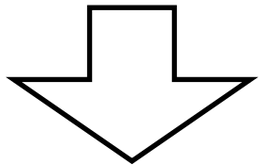
Maps /app/books to BookController.list()

Alternative syntax:

```
static mapping = {
    "/books" {
        controller = "book"
        action = "list"
    }
}
```

Parameterised mappings

```
"/books/$year"(controller: "book" , action: "list")
```

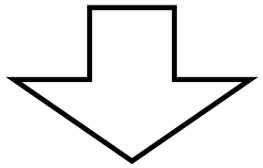


/app/books/2012 → BookController.list() + `params.year == "2012"`

/app/books → 404

Optional parameters

```
"/books/$year?"(controller: "book" , action: "list")
```

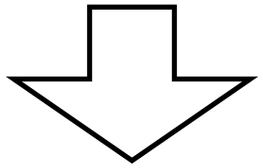


/app/books/2012 → BookController.list() + `params.year == "2012"`

/app/books → BookController.list() + `params.year == null`

Controller and action

```
"/$controller/$action?"()
```



/app/main/home → MainController.home()

/app/test → TestController.index()

Other mappings

Status code mapping:

```
"/401"(controller: "error", action: "notAuthed")
```

Mapping direct to a view (instead of an action):

```
"/about"(view: "about.gsp")
```

Relative to 'views' directory, i.e.
grails-app/views/about.gsp

Filters

What are filters?

- Request interceptors
  - Apply same processing to multiple request paths
- A Grails *artifact*
  - under grails-app/conf
  - must have a `Filters` suffix
- Multiple filters classes are allowed

Example

```
package org.example

class AppFilters {

    static filters = {

        logging(controller: '*', action: '*') {

            before = {
                log.info "Entering action"
                return true
            }
        }
    }
}
```

Required property

Controller/action pattern - any matching request will trigger this filter set

Name of this filter set - can have multiple of these

Returning `false` will stop the request processing

Filter types

Fired before the action executes:

```
before = {
    log.info "Entering " + controllerName +
             " - " + actionName
    return true
}
```

Implicit variables available in filters

Fired after the action executes:

Contains the view's model

```
after = { model ->
    log.info "Leaving " + controllerName +
             " - " + actionName
}
```

Filter types

Fired after the response is generated

Set to any exception that was thrown during processing - may be `null`

```
afterView = { Exception e ->
    log.info "Finished " + controllerName +
             " - " + actionName
}
```

Implicit variables

- In addition to `controllerName` and `actionName`:
  - `params`
  - `flash`
  - `session`
  - `request`
  - `response`

Summary

- URL mappings allow you to decouple URLs from controller actions
- Map error status codes to custom pages
- Use filters to intercept requests
  - for cross-cutting concerns such as profiling

GORM Mapping

What is it?

- Originally Grails Object Relational Mapping
- Maps objects to databases
- Uses conventions and sensible defaults
- Multiple implementations
  - Hibernate for relational databases
  - MongoDB
  - and more

Main parts

- Class mapping
  - How classes map to a relational schema
  - Or MongoDB document
- CRUD
  - Creating, updating, deleting and fetching records

Domain classes

- Persistent classes
  - Instances correspond to table rows or documents
- Another Grails artifact
  - In grails-app/domain
  - No class name suffix!

Basic mapping

```
class Book {
    String  title
    String  author
    Integer yearPublished
}
```

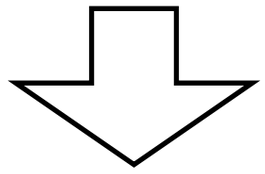


Table name from class name

Added by Grails

Properties are *not* nullable by default

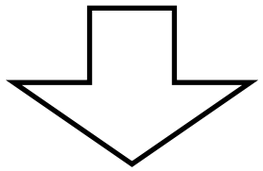
Column name from property name

```
CREATE TABLE book(
    id                  BIGINT,
    version             BIGINT,
    title               VARCHAR(255) NOT NULL,
    author              VARCHAR(255) NOT NULL,
    year_published      INTEGER NOT NULL,
    PRIMARY KEY (id)
);
```

Camel-case to underscore separated

Many-to-one

```
class Book {
    String  title
    Author  author
    Integer yearPublished
}
```



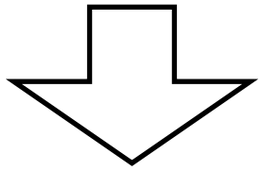
```
CREATE TABLE book(
    ...
    title              VARCHAR(255) NOT NULL,
    author_id          BIGINT NOT NULL,
    year_published     INTEGER NOT NULL,
);
```

<classname>_id
- foreign key

Unidirectional

Many-to-one

```
class Author {
    String name
}
```



```
CREATE TABLE author(
    ...
    name      VARCHAR(255) NOT NULL
);
```

Unidirectional

Many-to-one

Initialisation:

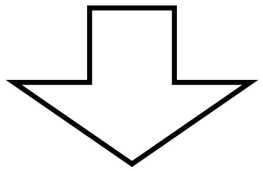
```
def author = new Author(name: "Ruby Wax")
author.save()

def book = new Book(
    title: "Sane New World",
    author: author,
    yearPublished: 2013)
book.save()
```

Relation must be saved first!

One-to-many

```
class Book {
    String  title
    Author  author
    Integer yearPublished
}
```



No change!

```
CREATE TABLE book(
    ...
    title            VARCHAR(255) NOT NULL,
    author_id        BIGINT NOT NULL,
    year_published   INTEGER NOT NULL,
);
```

Bidirectional

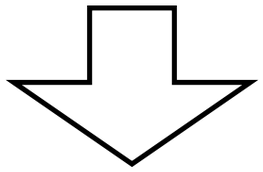
One-to-many

```
class Author {
    String name

    static hasMany = [books: Book]
}
```

Creates a books property in Author

No change to schema!



```
CREATE TABLE author(
    ...
    name      VARCHAR(255) NOT NULL
);
```

Bidirectional

One-to-many

Initialisation:

Named after the collection: 'books'

```
def author = new Author(name: "Ruby Wax")
author.addToBooks(new Book(
    title: "Sane New World",
    author: author,
    yearPublished: 2013))
author.save()
```

Don't need to save the book first - cascading save

Iteration:

```
for (Book book in author.books) {
    ...
}
```

One-to-many

```
class Book {
    String  title
    Integer yearPublished

    static belongsTo = [author: Author]
}
```

The type of the author property

Cascades deletes, i.e. deleting the author deletes the books too

Adds an author property to the Book class

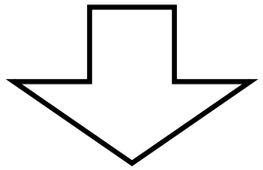
One-to-one

```
class Author {
    String name

    static hasOne = [profile: Profile]
    static hasMany = [books: Book]
}
```

Creates a profile property in Author

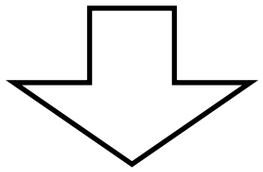
No change to schema!



```
CREATE TABLE author(
    ...
    name        VARCHAR(255) NOT NULL
);
```

One-to-one

```
class Profile {
    String fullName
    Date   dateOfBirth
    Author author
}
```



```
CREATE TABLE profile(
    ...
    fullName        VARCHAR(255) NOT NULL,
    date_of_birth   TIMESTAMP NOT NULL,
    author_id       BIGINT
);
```

Foreign key

One-to-one

Initialisation:

```
def author = new Author(
    name: "Ruby Wax",
    profile: new Profile(
        fullName: "Ruby Wax",
        dateOfBirth: ...))
author.save()
```

hasOne means we don't need to save the relation first

hasOne implies a `belongsTo` on the other side

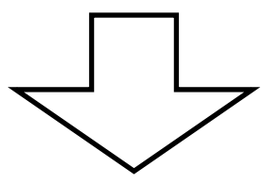
Many-to-many

```
class Book {
    String  title
    Integer yearPublished

    static hasMany = [authors: Author]
    static belongsTo = Author
}
```

Can be a list of classes for multiple belongsTos

One side *must have* a belongsTo



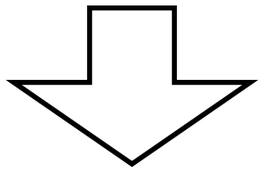
No author_id!

```
CREATE TABLE book(
    ...
    title            VARCHAR(255) NOT NULL,
    year_published   INTEGER NOT NULL,
);
```

Many-to-many

```
class Author {
    String name

    static hasMany = [books: Book]
}
```



```
CREATE TABLE author(
    ...
    name        VARCHAR(255) NOT NULL
);
```

No book_id!

Many-to-many

Named after the owning class, i.e. the class *without* the `belongsTo`

Named after the collection in `Author: books`

```
CREATE TABLE author_books(
    author_id    BIGINT NOT NULL
    book_id      BIGINT NOT NULL
);
```

Join table!

Many-to-many

Initialisation:

Named after the collection: 'books'

```
def author = new Author(name: "Ruby Wax")
author.addToBooks(new Book(
    title: "Sane New World",
    author: author,
    yearPublished: 2013))
author.save()
```

Don't need to save the book first - cascading save due to belongsTo

NOT:

```
def book = new Book(
    title: "Sane New World",
    yearPublished: 2013)
book.addToAuthors(new Author(name: "Ruby Wax"))
book.save()
```

hasMany

- Not recommended for large collections
  - All elements must be loaded for insertion
- Deleting and removing elements difficult
  - Often foreign key constraint violations

belongsTo

- Only defines cascading behaviour
- Multiple variations
  - a class literal
  - a list of class literals
  - a map of property names and classes
- Map syntax may affect schema due to the concrete property it adds

Special properties

```
class Book {
    String  title
    Integer yearPublished
    Date    dateCreated
    Date    lastUpdated

    static belongsTo = [author: Author]
}
```

Automatically set whenever a new instance is first saved

Automatically set whenever a new or existing instance is saved

These are *auto-timestamp* properties

Demo

Dynamic Scaffolding

Domain class to scaffold

```
class AuthorController {
    static scaffold = Author

// or static scaffold = true
}
```

Scaffolds the domain class with the same base name as this controller

This does not make it static scaffolding

- Shortcut command:
  - `create-scaffold-controller org.example.Author`
- UI reflects changes in domain classes
  - i.e. UI is always up to date

Static scaffolding

- Created with `generate-all`
- Creates fully populated controller and views
  - allows you to see the real code
- Doesn't update with the domain classes
- Only useful as a self-learning tool

Demo

Embedded mapping

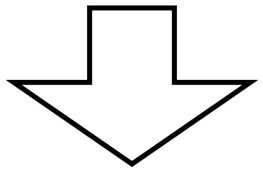
```
class Address {
    String number
    String city
    String postCode
}
```

Put class in src/groovy

```
class Author {
    Address address

    static embedded = ["address"]
}
```

List of property names



```
CREATE TABLE author(
    ...
    address_number      VARCHAR(255) NOT NULL,
    address_city        VARCHAR(255) NOT NULL,
    address_post_code   VARCHAR(255) NOT NULL,
);
```

Inheritance

```
class User {
    String username
    String passwordHash
}
```

```
class Employee
        extends User {
    String employeeId
    String division
}
```

```
class Customer
        extends User {
    String  customerId
    Company company
}
```

Domain classes

Inheritance

```
CREATE TABLE user(
    username        VARCHAR(255) NOT NULL,
    password_hash   VARCHAR(255) NOT NULL,
    class           VARCHAR(255) NOT NULL,
    employee_id     VARCHAR(255),
    division        VARCHAR(255),
    customer_id     VARCHAR(255),
    company_id      BIGINT
);
```

Properties in sub-classes must be nullable

Table per hierarchy schema (the default)

Inheritance

```
CREATE TABLE user(
    username        VARCHAR(255) NOT NULL,
    password_hash   VARCHAR(255) NOT NULL
);
```

```
CREATE TABLE employee(
    user_id         BIGINT NOT NULL,
    employee_id     VARCHAR(255) NOT NULL,
    division        VARCHAR(255) NOT NULL
);
```

```
CREATE TABLE customer(
    user_id         BIGINT NOT NULL,
    customer_id     VARCHAR(255) NOT NULL,
    company         VARCHAR(255) NOT NULL
);
```

Table per class schema

Activating table-per-class

```
class User {
    String username
    String passwordHash

    static mapping = {
        tablePerHierarchy false
    }
}
```

- Requires a JOIN for most queries
- Requires extra tables
- Can have `nullable: false` for sub-class properties
- More space efficient

Summary

- GORM maps classes to relational tables
  - by convention
  - with customisation possible
- Only domain classes are mapped and persisted
- Various association types supported
  - Many-to-one with direct property reference
  - One-to-many with `hasMany`
  - Many-to-many with `hasMany` + `belongsTo`

Summary

- Uni-/bidirectionality depends on existence of properties
- `belongsTo` only defines cascading behaviour
- Scaffolding provides basic CRUD UI for domain classes
- Dynamic scaffolding updates with changes

GORM CRUD

CRUD

Persisting domain objects

```
def author = new Author(name: "Ruby Wax").save()
```

Persists the object to the database

- save()
  - performs validation
  - if validation passes
    - persists the object to the database
    - returns the saved object
  - otherwise returns `null`
- validate() just performs the validation

Validation

```
class Profile {
    String fullName
    Date   dateOfBirth
    String email

    static belongsTo = [author: Author]

    static constraints = {
        fullName    blank: false, nullable: false
        dateOfBirth max: new Date()
        email       blank: false, email: true
    }
}
```

Constraints block like for command objects

- Like command objects, domain objects have
  - An `errors` property (`org.springframework.validation.Errors`)
  - A `hasErrors()` method

Other effects

- `nullable: false` is the default for domain classes
  - Marks column as `NOT NULL`
- `maxSize` and `size` constraints affect SQL type for strings
  - e.g. a large size may switch from VARCHAR to TEXT

Data Binding

In an action:

```
class BookController {
    def update(Book book) {
        if (book.validate() && book.save()) {
            ...
        }
    }
}
```

- Grails binds `params` to the action argument
- *All* parameters are bound that have matching properties
  - Insecure!

Data Binding

In an action:

```
class BookController {
    def create() {
        def book = new Book(params)
        if (book.validate() && book.save()) {
            ...
        }
    }
}
```

- *All* parameters are bound that have matching properties
  - Insecure!

Data Binding

In an action:

```
class BookController {
    def create() {
        def book = new Book()
        bindData(book, params, [include:
                ["title", "author.id"]])

        if (book.validate() && book.save()) {
            ...
        }
    }
}
```

- Binding with a white list - secure!

The R & the D

- Every domain class has a *static* `get()` method
  - e.g. `def book = Book.get(10)`
  - Only works if you know the ID
  - Uses the 2nd-level cache if configured
- Every domain object has a `delete()` method
  - Removes the associated database record
  - May fail due to referential integrity violations

Querying

Creating initial data

grails-app/conf/BootStrap.groovy:

```
import org.example.*

class BootStrap {
    def init = {
        def author = new Author(name: "Ruby Wax")
        author.addToBooks(new Book(
            title: "Sane New World",
            author: author,
            yearPublished: 2013))
        author.save(failOnError: true)
    }
}
```

Closure property called on application startup

Throws exception if validation fails

Where queries

The domain class/table we're querying on

Criteria specified as Groovy expression

```
def query = Book.where {
    title == "Colossus"
}

query.get()
```

A property on Book - must be on left hand side!

```
def query = Book.where {
    title == "Colossus"
}

query.list()
```

Fetches all matching instances of Book

Operators

Operator	SQL	Description
==	=	Equality
<	<	Less than
>	>	Greater than
!=	<>	Not equal
>=	>=	Greater than or equal
<=	<=	Less than or equal
in	in	In a list of values
==~	like	String pattern match
=~	ilike	Case-insensitive pattern match

```
    IS NULL => prop == null
IS NOT NULL => prop != null
```

```
BETWEEN => prop in 0..100
```

Compile-time checks

```
def query = Book.where {
    tite =~ "C%"
}

query.list()
```

Unknown property on Book - compiler error!

```
def query = Book.where {
    title <=> "Colossus"
}

query.list()
```

Unsupported operator - compiler error

Combining criteria

Query on associations

```
def query = Book.where {
    title =~ "C%" || author.name == "Niall Ferguson"
}

query.list()
```

Corresponds to SQL 'OR' -
&& supported too

```
def author = ...
Book.where {
    title =~ "Colossus"
    if (author) {
        author.name == author
    }
}.list()
```

Criteria on separate lines implicitly ANDed

On right-hand side, so refers to local variable

Selecting properties

```
def query = Book.where {
    title =~ "C%"
}.property("title")

assert query.list() == ["Colossus", "Collapse"]
```

Single property results in list of values for that property

```
def query = Book.where {
    title =~ "C%"
}.property("title").property("yearPublished")

assert query.list() == [
        ["Colossus", 2003],
        ["Collapse", 2005] ]
```

Multiple properties result in list of lists

Advanced querying

- HQL is like SQL
  - Uses properties and classes rather than columns and tables
- Can return domain instances or lists of values
- Supports JOINs, GROUP BY, aggregate functions, etc.

Basic HQL

Returns domain instances

```
def books = Book.findAll(
    "from Book where title like 'C%'")
```

Required - must match the class you're calling `findAll()` on

Returns list of values

```
def titles = Book.executeQuery(
    "select title from Book where title like 'C%'")
```

Joins

```
def profiles = Profile.findAll(
    "from Profile p join p.author a where a.name" +
    " like 'Niall%'")
```

Association to join

Alias for the association

```
def results = Book.executeQuery(
    "select b.title, a.name from Book b " +
    "join b.authors a where b.title like 'C%'")
```

Aggregates

Aggregate function

```
def results = Author.executeQuery(
    "select a.name, count(b) from Author a " +
    "join a.books b group by a.name")
```

SQL-like GROUP BY clause

What SQL is executing?

- Add dataSource.logSql = true to DataSource.groovy

```
dataSource {
    pooled = true
    driverClassName = "org.h2.Driver"
    username = "sa"
    password = ""
    logSql = true
}
```

Demo

Other query options

- Dynamic finders
  - e.g. `Book.findAllByTitleLike("C%")`
  - Can't query on properties of associations
  - No compile-time checks
- Criteria API
  - Similar power to HQL
  - A DSL rather than SQL-like, but a little confusing
- See Grails User Guide for info

Summary

- CRUD covered by `get()`, `save()` and `delete()`
- Initial data can be provided in BootStrap.groovy
- Multiple query options
  - Where queries and HQL preferred
  - Dynamic finders and criteria API if you want
- Log SQL to diagnose database load

Spring and Transactions

Dependency Injection

A "collaborator" object

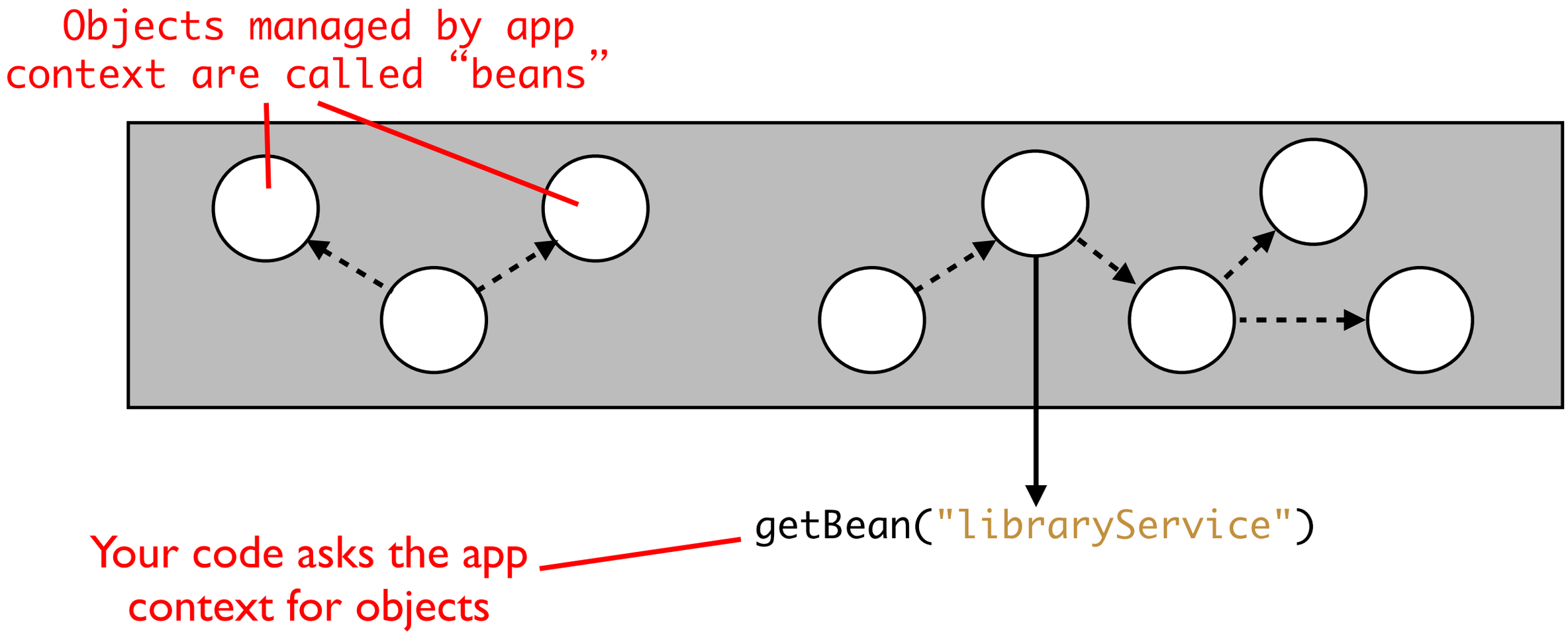
```
class BookController {
    def libraryService

    def list() {
        [books: libraryService.allBooks()]
    }
}
```

How does this get initialised?

Our code doesn't instantiate a LibraryService object, so why does the above work?

Application Context



- Application context instantiates objects and wires them together
- The wiring is called "dependency injection"

Dependency Injection benefits

- No plumbing code in your classes
- Easy testing - inject mocks
- Use alternative implementations based on environment

Auto-wiring

- All services are beans
- Bean name == property version of class name
  - e.g. `LibraryService` has the name "libraryService"
- Properties with same name as a bean will be injected with that bean

```
class BookController {
    def libraryService

    def list() {
        [books: libraryService.allBooks()]
    }
}
```

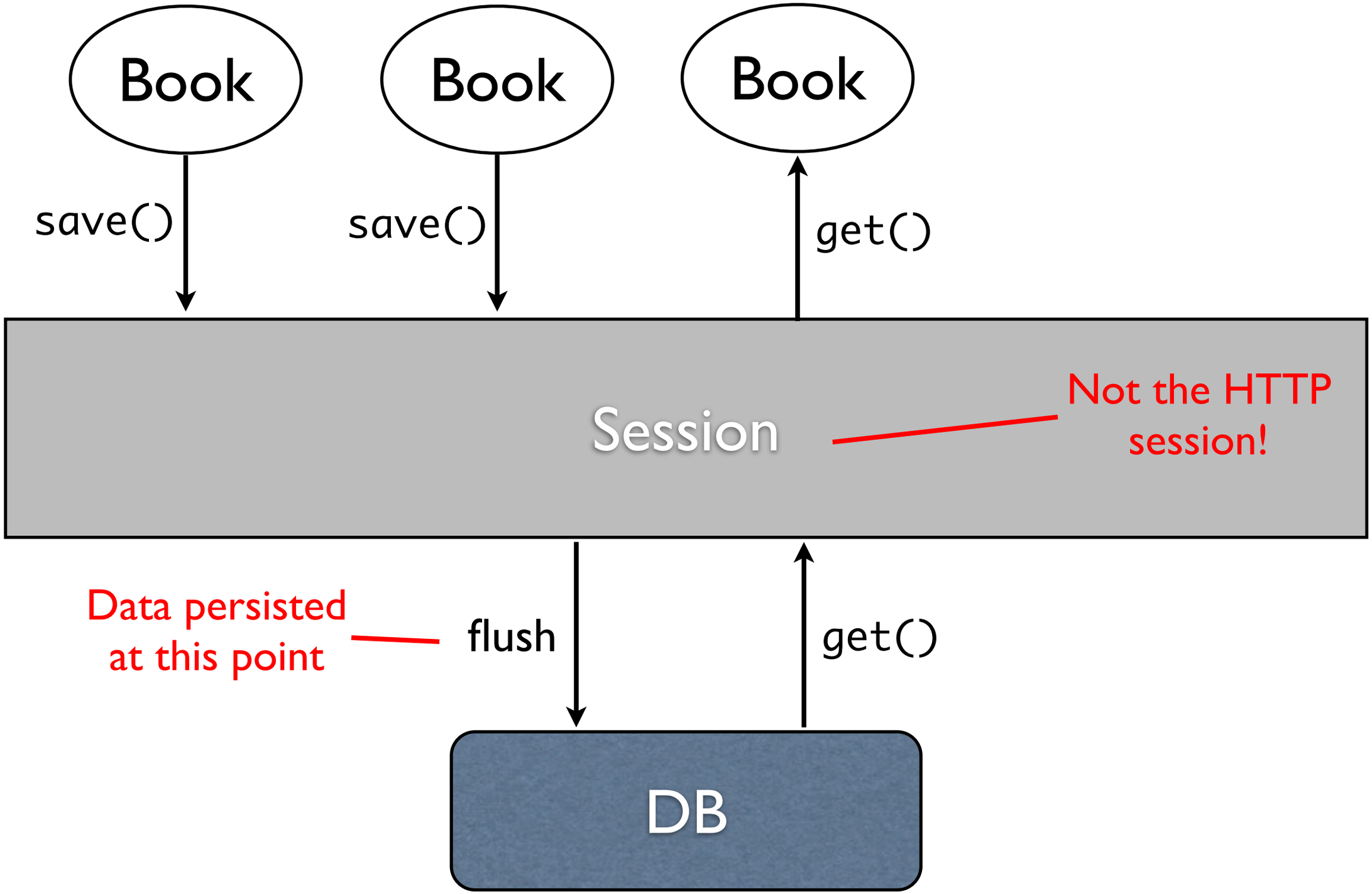
Injected with instance of `LibraryService` on construction

Spring

- Dependency injection is core of Spring
  - `org.springframework.context.ApplicationContext`
- You can define your own beans
  - grails-app/conf/spring/resources.groovy
  - grails-app/conf/spring/resources.xml
- Spring provides the transaction support in Grails

Persistence Session

save != persist



Persistence Session

- Alternatively called Hibernate Session
- Acts as a cache
  - Key-value store of IDs to domain objects
- Flush may happen at any time unless forced:
  - `domainObject.save(flush: true)`
  - `domainObject.delete(flush: true)`
- By default, a session is open for the entire life of an HTTP request

Persistence Session

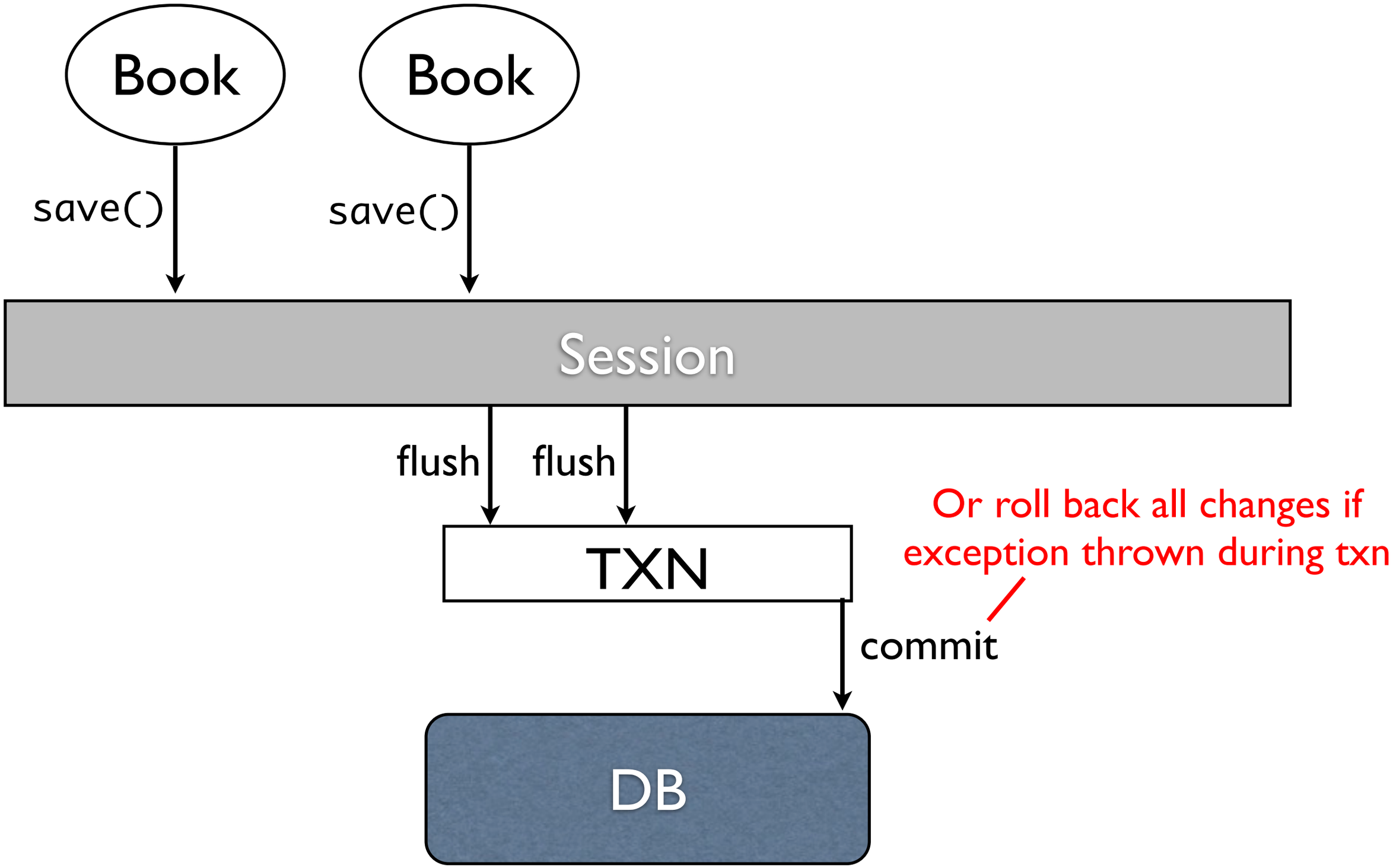
- Flush may happen at any time unless forced:
  - `domainObject.save(flush: true)`
  - `domainObject.delete(flush: true)`
- A flush always happens when session is closed
- Any changes to objects in the session are persisted on flush - *even if there is no* save()
- A session is *not* a transaction

Demo

Transactions

- Guarantee ACID
  - Atomicity, Consistency, Isolation, Durability
- Focus on units of work
  - All changes persist, or none
- Enforced at the database level
  - A SQL COMMIT required

Transactions



Transactions in Grails

This is the default for services

```
class LibraryService {
    static transactional = true

    def createAuthor(
            String name,
            String emailAddress,
            Date dateOfBirth) {
        ...
    }
}
```

A transaction wraps every public method

End of method commits transaction, or rolls it back on exception

Transaction attributes

Grails 2.3+ - for earlier Grails,
org.springframework.transaction.annotation.Transactional

```
import grails.transaction.Transactional

class LibraryService {
    static transactional = true

    @Transactional(readOnly=true)
    def allBooks() {
        ...
    }
}
```

Configure transactions through an annotation

Transaction and exceptions

```
@Transactional
def createAuthor() {
    ...
    throw new IOException()
}
```

Checked exception - will *not* roll back transaction

```
@Transactional
def createAuthor() {
    ...
    throw new RuntimeException()
}
```

Runtime exception - *will* roll back transaction

Fine-grained transactions

TransactionStatus object
(org.springframework.transaction)

```
class LibraryService {
    static transactional = false

    def createAuthor(...) {
        Author.withTransaction { status ->
            ...
            if (notSuccessful) {
                status.setRollbackOnly()
            }
        }
    }
}
```

This can be literally
any domain class

Roll back the transaction
even with no exception

Summary

- Spring manages object creation and wiring
  - through the application context
- Grails defaults to auto-wiring by name
- GORM data access goes through session
  - not transactional!
- Transactions through service methods
- Runtime exceptions roll back