Services

Where the magic happens

Section goals

After this section, you will be able to

- Encapsulate reusable business logic into service methods
- Wrap database transactions around service code
- Write unit and integration tests for service classes

Introduction

- Services are classes for centralizing the business logic in your application
  - Also a great place for all of your domain queries/updates
- Service class is a POGO (Plain Old Groovy Object) - no special superclasses to extend or interfaces to implement
- Services live in the `grails-app/services` folder

```
class MyService {
  def doSomething() {
    // magic goes here ...
  }
}
```

Service availability

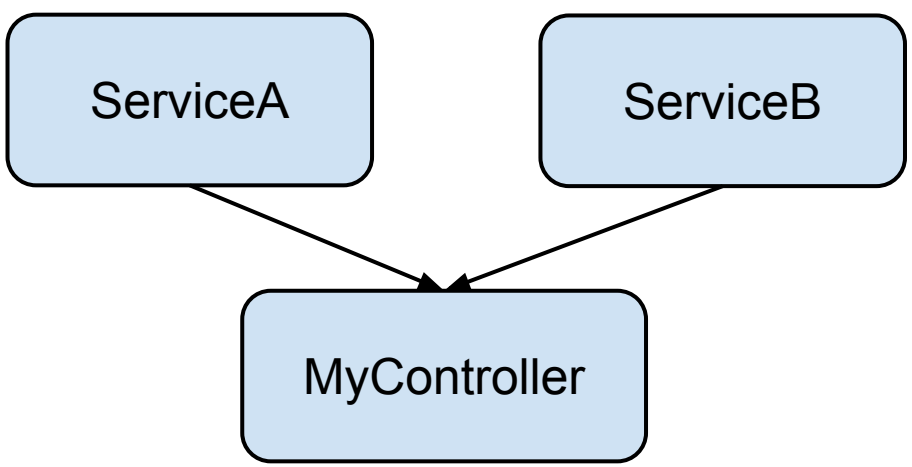
Service classes can easily be used from controllers, taglibs, even other services

- Convenient place to centralize common code

Available through dependency injection

Dependency Injection - Introduction

- Dependency Injection (DI) - classes have their collaborators ('dependencies') given to them by the container (Spring)



Dependency Injection - Advantages

- Easier to test
  - Easily use mock or stub dependency in unit test
- Loose coupling
  - Class isn't dependent on underlying implementation of dependency

Dependency Injection - Services

- Grails services are automatically available for injection into controllers, domain classes, filters, taglibs, conf classes, and even other services

```
class MyController {
  def myService  //Instance of MyService injected by Grails

  def myAction() {
    myService.doSomething()
  }
}
```

Service injection - syntax

Can specify injected dependencies as weakly-typed 'def'

```
class MyController {
  def myService // Injects service class 'MyService'
}
```

Or strongly-typed dependency

```
class MyController {
  MyService myService
}
```

The key is the name of the service field - it **must** be the service class name in camel-case (with first letter lowercase)

Service injection - more examples

Injection into Taglib classes:

```
class MyTagLib {
  def myService

  def getMyValue() {
    out << myService.caculateValue()
  }
}
```

Injection into other services:

```
class MySecondService {
  def myService
}
```

Services - naming caveat

- Dependency injection fails if the service class name starts with two capital letters
  - Ex: class EDocumentService

```
class MyController {
  def eDocumentService // Won't get injected
  ...
}
```

- Use a service class name like DocumentService instead

- Convention is to end service class names with 'Service'

Services - singleton and stateless

- Service instances that are injected are singletons
  - Instances may be used concurrently

- Consequently, services must be stateless
  - No member variables or any other kind state

```
class MyService {
  String status // Don't do it!
  def myServiceMethod() {
    if (status == "new") {
      status = "started"
    }
  }
}
```

Services - default transactions

By default, each service method runs inside a transaction

```
class MyService {
  def doSomething() {
    User newUser = new User()

    newUser.save()

    UserRole newUserRole = new UserRole(user: newUser)

    // If this fails, 'newUser' gets rolled back as well
    newUserRole(failOnError: true)
  }
}
```

Services - disable transactions

Can specify that no methods in the service should be inside a transaction:

```
class MyService {
  static transactional = false

  // No transaction
  def method1() {
  }

  // No transaction here either
  def method2() {
  }
}
```

Services - @Transactional

- Only want a transaction around certain methods in your service?
  - @Transactional
- Also disables transactions for other methods in the service

```
class MyService {
  @Transactional
  def methodWithTransaction() {
  }

  def methodDoesNotHaveTransaction() {
  }
}
```

Services - programmatic transactions

For times when you want tighter control over the scope of a transaction: `.withTransaction {}` closure

```
class MyService {
  static transactional = false

  def serviceMethod() {
    def users = User.list()
    users.each { user ->
      User.withTransaction {
        // User update code
      }
    }
  }
}
```

Transaction rollback

Transaction rollback simple with programmatic transactions

```
class MyService {
  static transactional = false

  def serviceMethod() {
    def users = User.list()
    users.each { user ->
      User.withTransaction { status ->
        // User update code
        status.setRollbackOnly()
      }
    }
  }
```

Creating services

Run Grails command

```
> grails create-service <package_name>.<service_class_name>
```

For example

```
> grails create-service com.opi.MyService
```

Creates

- `grails-app/services/com/opi/MyService.groovy`
- `test/unit/com/opi/MyServiceTests`

Services - integration tests

Integration tests live in the `test/integration` folder

Services are injected into integration test classes

```
@TestFor(VotingService)
class VotingServiceSpec extends Specification {
    def votingService

     void "test something"() {
     }
}
```

Integration tests - transaction

Each integration test method runs inside a transaction that is rolled back at the end of the test case

- Reverts domain changes so test methods don't affect each other

Can disable transactions in integration test just like service

```
static transactional = false
```

Workshop

Create a domain object, com.opi.Vote

```
class Vote {
    static final UP_VOTE = 1
    static final DOWN_VOTE = -1

    User user
    Integer vote

    static constraints = {
        vote validator: { it in [DOWN_VOTE, UP_VOTE] }
    }
}
```

Workshop

Both Question and Answer domains can have many votes

```
static hasMany = [votes:Vote]
```

Workshop

- Create a VotingService in the com.opi package
- Add method with signature

```
Vote addVoteToQuestion(Question question, User votingUser, Integer vote)
```

- And method with signature

```
Vote addVoteToAnswer(Answer answer, User votingUser, Integer vote)
```

- Bonus Points for only allowing the user to vote 1 time

Workshop - use service

- Update the `VoteController` to use the new `VotingService` for adding votes to `Questions` and `Answers`
- Add a `def votingService` field to `VoteController` to inject the service
- Use User with the id of 1

Custom injected classes

Can define your own classes that are injected just like services

grails-app/conf/spring/resources.groovy

```
beans = {
  // beanName(BeanClass)
  myClass(MyClass)
}

class MyController {
  def myClass
}
```