An Evaluation Based on Firsthand Implementation Experience

Rajeshkumar I.P
Grails Web Framework: An Evaluation Based on Firsthand Implementation Experience

Contents

1. Executive Summary 3

2. Getting to Know Grails 3
   2.1. Putting Grails to the Test 4
       2.1.1 Consideration #1: Ramp-up and Set-up 4
       2.1.2 Consideration #2: Scaffolding and Template Support 5
       2.1.3 Consideration #3: Java Interoperability 6
       2.1.4 Consideration #4: Validation Support 7
       2.1.5 Consideration #5: Plugins and Tag Lib Support 8
       2.1.6 Consideration #6: GORM 9

3. Design Considerations 9
   3.1.1 Best Practices 9
   3.1.2 Design Patterns 11
   3.1.3 Issues and Pitfalls 12-13

4. Benefits of Using Grails 14
   4.1. Time to Market 15
   4.2. Cost Reduction 15

5. When to Use Grails 16-17

6. Comparison and Adoption 17-18

7. Grails and Groovy: Looking to the Future 19

8. Conclusion 20

9. References 20
1. EXECUTIVE SUMMARY

The market is awash in Java web frameworks, each claiming to be better than the next. The goal of this paper is not to assess which framework is superior. Rather, the intent is to share firsthand experience in using one particular framework: Grails and its programming language, Groovy. To that end, this paper discusses the framework in the context of real-world application—highlighting advantages, challenges and pitfalls, as well as exploring the framework’s long-term potential.

2. GETTING TO KNOW GRAILS

Grails is a next-generation Java web development framework that generates great developer productivity gains through the confluence of a dynamic language, a convention over configuration philosophy, powerfully pragmatic supporting tools, and an agile perspective drawn from the best emerging web development paradigms.

Grails is an Open Source, full stack, web application framework for the Java Virtual Machine (JVM). It takes advantage of the Groovy programming language and convention over configuration to provide a productive and streamlined development experience.

Grails is a full stack framework that attempts to solve as many pieces of the web development puzzle through the core technology and its associated plugins. Grails as a framework has not reinvented the wheel, but instead has leveraged the already tried and trusted frameworks such as Spring, Hibernate, Sitemesh and other open source libraries already popular in the enterprise Java world. Grails’ model-view-controller (MVC) is an abstraction built over Spring MVC. The Grails persistence framework, GORM, is an abstraction built over Hibernate.

Implementing most Java web frameworks can be tedious—requiring extensive time to edit configuration files, customize web context files, write injection definitions, modify page layouts, build scripts and restart apps following each change. These tasks consume the majority of the developer’s time at the expense of tackling the real business problem and functional requirements.

By contrast, Grails offers the best of the available Java web frameworks while shielding the developer from the noisy configuration, design complexity and boilerplate code that make existing Java web development so cumbersome and inefficient. In short, Grails allows the developer to focus time implementing features, not editing XML, while also providing a productive, streamlined development experience.
2.1. Putting Grails to the Test

An investment banking company engaged Sapient Global Markets to develop a reference data application in support of its commingled investment vehicles (funds/pools). The client required an application with these high-level features:

- Intuitive and simple set-up to maintain reference data in terms of structure and attributes with View/Add/Edit functionality.
- Integration and interoperability with legacy Java Struts 1.x application, deploying them together as one application.
- Dynamic validation and mandatory check for fields/attributes driven by specific rules.
- Dynamic fetching of attribute values from different applications on pre-conditions.
- Audit/history functionalities.
- Search, change status and notifications.
- Entitlements at each field and page level depending on roles and responsibilities.

The Sapient team took a close look at Grails—addressing a number of considerations and requirements and ultimately determining that it would be a viable Java web framework for this client application.

2.1.1 Consideration #1: Ramp-up and Set-up

Key requirement:
“The team has no experience with Grails. Won’t a lot of time be lost in ramping up and getting the application wired?”

All the developers on the team have a pure Java web development background, so this represented their first experience working on Grails. Even so, ramping up on Grails did not involve a huge learning curve. The Sapient developers were able to ramp up on the Grails framework and Groovy language easily and quickly:

- The Grails core is built on the already proven Spring MVC and Hibernate framework. Thus, the developers found it easy to understand the “nuts and bolts” of the framework.
- As an object-oriented programming language, Groovy is similar to Java; Java developers were able to learn the language easily.

Since Grails makes use of convention over configuration, it generates all the boilerplate shell code. It thereby avoids all unnecessary XML configuration and design complexity developers face when using other Java web frameworks, such as Spring MVC. With Grails, Sapient’s developers were up and running—and productive—quickly.
2.1.2 Consideration #2: Scaffolding and Template Support

Key requirement:
“We have around 80 Create/Edit/View pages in the application, with approximately 30 to 40 attributes/fields per screen. How will we ensure we are not repeating the effort so we can design these pages more efficiently?”

Grails offers "scaffolding," which lets the developer auto-generate views and controller actions for Create/Read/Update/Delete (CRUD) for any given domain class. Grails provide two types of scaffolding: dynamic and static.

**DYNAMIC SCAFFOLDING**
Dynamic scaffolding is achieved by enabling the scaffold property to “true” in the controller (assuming that the controller follows the same naming convention as the domain class). In cases where the domain names are different, it is necessary to specify the name of the domain to the scaffold property.

```java
class POVController {
    static scaffold = true
}
```

With this configuration, if the application is now started, it will auto-implement all the CRUD-related actions within the controller and also generate the respective views.

**STATIC SCAFFOLDING**
With static scaffolding, Grails allows the developer to generate a controller and the views used to create the user interface from the command line.

This can be achieved by running the “`grails generate-* <domain>`” command.

Both of these use the default templates, which come bundled with Grails, to generate the view and controllers for a particular domain. Static scaffolding is beneficial when working to build a prototype, test an idea or create an admin interface.

In a real-world scenario, the default scaffolding templates would not suffice as each application would demand a specific structure and look and feel for the user interface. Fortunately, Grails remains true to its don’t-repeat-yourself (DRY) philosophy, providing an ability to customize the default scaffolding templates.

Some of the steps required for customizing templates include:
1. Run the “install-template” command.
2. This copies the templates Grails uses for all code generation to the application’s “src/templates” directory.
3. You can then customize the copied template directly or make a copy of it and customize the copy per your requirement.

Grails also provides an excellent plugin called “[Spawn plugin](#)” that allows you to denote a specific template for a domain class. This way you can have multiple templates within your application and assign them to whichever domain class you would like to use them with for scaffolding.

In this case, the developers created three customized scaffolding templates—one for Create, another for View and the third for Edit/Update—and generated respective views and controllers for a given domain using the spawn plugin.

Doing so ensured a consistent structure, look and feel, easy maintainability for all the user interface/views across the application, and saved about 60 percent of development time—hours that otherwise would have been spent writing it by hand.
2.1.3 Consideration #3: Java Interoperability

Key requirement:
“We already have an existing application developed in Struts 1.x. Although it is currently deployed as a separate application, we now need it to coexist with the new reference data application. Can we integrate the applications without migrating the legacy application to the Grails framework?”

This challenge was complex, introducing the following requirements:

- Both the Struts and Grails frameworks would have to coexist as a single application.
- It would be bundled as one Web Application Archive (WAR) and deployed as one application on a Tomcat web container.
- Depending on the request URI, the web container would route the request to either of the frameworks and then load the respective pages.
- The integrated application would also need to be capable of invoking Java Service classes in the Struts application while performing Create/Update operations from the Grails application.

A key advantage of Grails is the ability to intermix Java and Groovy code in the same file. The code works seamlessly and is eventually converted to JVM-specific byte code. Thus, it became easier to integrate the already written Java files and services into the Grails framework.

Additionally, Grails encourages plugins, which helps build modularity into an application. It thereby promotes DRY principles and encourages reusability. The Grails community provides many of common plugins (Security, Caching and ORM, to name a few).

Grails provides a Struts1 plugin, which Sapient developers used to bundle the Struts and Grails applications. Further, the team was able to render either the Struts or Grails application with the help of URL filters. Able to complete this integration very easily and efficiently, the team saved a great amount of time and effort that would not have been avoided if using any other Java framework.

What follows are some of the steps required as part of the integration:

1. Copy Struts Java source files from `<struts-app>/source/java` to `<grailsProject>/src/java`
2. Copy all folders (except WEB-INF & META-INF) from `<Struts-app>/source/WebApplication` into `<grailsProject>/web-app`
3. Copy all TLDs and XMLs (except web.xml) from `<Struts-App/source/WebApplication/WEB-INF` into `<grailsProject>/web-app/WEB-INF`

Copy all jars from `<Struts-App/source/WebApplication/WEB-INF/lib` into `<grailsProject>/lib`
2.1.4 Consideration #4: Validation Support

Key requirement:
“The mandatory fields on specific screens would not be fixed. They would be dynamic depending on a set of conditions, such as Status or Line of Business. This set could vary in the future. How can we provide all the field-specific validations—today and tomorrow?”

Built into the Springs Validator API and data binding capabilities, Grails validation provides a unified way to define validation constraints. Grails provides a number of out-of-the-box validators while also enabling the developer to create custom validators.

What follows is how a typical constraint would be defined in Grails—referring to a domain example where some of the out-of-the-box and custom validators are used.

```groovy
class GrailsPov {
    ...
    static constraints = {
        fundName size: 5..35, blank: false, unique: true
        baseCurrency(nullable: true, IsRequiredWhen: [[ATTRIBUTE_NAMES]: [MANDATORY, STATUS]]
    }
}
```

However, in the test scenario, the Sapient team wanted to provide dynamic validations. This is where Grails’ custom validator feature came in handy. What follows is an explanation of what was achieved:

1) Create a custom validator class extending “AbstractConstraint” class and overriding the “processValidate()” method.

2) Once the custom validator is created, it should be registered in the Config.groovy file so that it is available globally and hence could be used in any of the domain or Custom classes.

3) If we assume custom constraint is called “IsRequiredWhenConstraint” it would be registered as follows:

```groovy
```

Grails also supports the concept of “Command Object” in situations where data validation may not fit into an existing domain class. One example: a scenario in which some of the custom validations logic uses a service class.

The Sapient team found Grails’ validation framework to be useful, providing ample flexibility for implementation.
### 2.1.5 Consideration #5: Plugins and Tag Lib Support

**Key requirement:**

"The audit needs to be done on all the entities to track changes. It is also important to be able to export data to Microsoft Excel and to a scheduled job that would run on a daily basis. The application must populate a staging table against which the dashboard would fetch the data. How can we ensure that the fields on the page are security enabled so that depending on user roles, they would be editable or disabled?"

With plugins, Grails supports reusability and modularization. The framework comes with a large set of common plugins, which the Sapient team used extensively to address the requirements listed above.

Use of plugins enhanced implementation efficiency and saved valuable development time:

- Grails includes an "Audit Logging Plugin," which provides Hibernate event-based logging as well as support for the "OnSave," "OnChange" and "OnDelete" closures. The team achieved audit logging by implementing "OnChange" and "OnSave" event handlers within all the domain classes. "OnChange" provides a signature wherein handlers to both the old and new state of the instance object can identify, log and compare exact changes.

- The "Excel-Export" Grails plugin, which provides the ability to convert data to a Microsoft Excel format, is built on top of Apache POI.

- Grails comes with a Quartz plugin, which the Sapient team leveraged to configure the scheduled job to populate data to the staging table. The integration was relatively smooth.

When it comes to tag libraries, Grails provides excellent support—simple and elegant functionality that completely reloads at runtime. The tag library helps in Groovy Server Page (GSP) rendering and fulfills the role of "view helper." Unlike Java Server Page (JSP), GSP makes it very easy to create tag libraries.

In Grails, a tag library is a class with the name that ends in the convention "TagLib" and lives in the "grails-app/taglib" folder. Below is the code to create a tag:

```java
class SimpleTagLib {
    def dateFormat = { attrs, body ->
        out << new java.text.SimpleDateFormat(attrs.format).format(attrs.date)
    }
}
```

For the security entitlements, the Sapient team created a logical tag library called "isAuthorized." That library was then used in GSP to control whether or not a field could be editable. Creating a tag ensured the code is reused, not repeated. It also provided good modularity and maintenance.

```xml
<g:isAuthorized user="${myUser}"/>
<g:textField name="pov" value="povInstance?.name"/>
</g:isAuthorized>
```

-Grails Web Framework: An Evaluation Based on Firsthand Implementation Experience-
2.1.6 Consideration #6: GORM

Key requirement:
“Hibernate needs to be used as the persistence layer for this application, which will deal with a lot of reference data to be fetched from various legacy applications. Is Grails well suited to be a 'lightweight' framework for that, as well?”

Grails provides a persistence layer called Grails’ Object Relational Mapping, or GORM, which is an abstraction over the Hibernate ORM framework. Because it is based on the convention over configuration paradigm, GORM eliminates all the tedious Hibernate configuration files—instead providing a “shield” with everything driven by domain.

GORM supports all the same relational data sources as Hibernate, as well as some of the non-relational data sources (Redis and Mongo DB, for instance).

The team decided to use GORM as the persistence layer for all of the core tables and domains for the application and for all of the legacy applications. The team chose iBATIS as the persistent framework, driving a great deal of efficiency and performance gains because native SQL queries were executed. It also gave a clear distinction on the persistence design strategy for the application.

One of Grails’ excellent features is the ability to bypass GORM and use Hibernate API directly to map POJO/POGO to existing tables.

Clearly, the Grails framework was able to provide the solution for all of the requirements detailed in the preceding section. In particular, Grails helped significantly simplify implementation around custom validations, cross-application integration and security entitlements. By contrast, other web frameworks, such as Spring or Struts, would have been far more complex to implement and maintain.

3. DESIGN CONSIDERATIONS
3.1.1 Best Practices

While there is no single best way to develop a Grails application, there are some guidelines that will lead to a more maintainable application with a scalable design and fewer bugs. The following section outlines some recommended practices every Grails developer should follow as part of the development.

VIEWS

- Develop reusable tag components for common parts of the UI.
- Refrain from adding any business or database logic in the view layer.
- Use convention-based layouts over explicit meta tags; creating a specific layout for one particular action enhances maintainability versus meta-magic branching. Additionally, it will ensure a consistent look across all application pages.
- For common HTML fragments or repeated content, put them in templates (for example, “_includeSection.gsp”). They can also be included in specific view pages using the render page tag.
- For larger datasets, apply pagination by making use of the “pagination” tag.
- Grails provides good support for JavaScript (JS) libraries (such as YUI, ExtJS and JQuery). Choose the right library for the application; opt for one that would provide the right AJAX and scripting support across the app. Ideally, stick to a single JS library rather than including multiple JS libraries.
- Select an appropriate layout library (such as YUI) when developing complex CSS layouts. Doing so helps ensure cross-browser compatibility and will drive efficiency.
- Minification and caching of static CSS and JS files are a good practice for improving performance of web pages. Leverage the Grails plugin in “yui-minify-resources” to achieve this. Handle Flash message display in the layout to display messages on the view screen rather than repeating it in each view.
3. DESIGN CONSIDERATIONS

3.1.1 Best Practices (continued)

**CONTROLLERS**

- Keep controllers “lean and clean” and ensure that business logic, queries, and updates are not performed within them.

- All the heavy lifting in terms of business logic and queries should be handled by a service class that provides a simple, maintainable and testable way of encapsulating reusable business logic.

- While performing redirects within controllers, use Flash scope and Flash message to display messages as they survive one redirect.

- Set default action in controllers by using the “defaultAction” command.

- Make use of the errors object on the domain class to display validation messages. Take advantage of resource bundles to make error messages relevant to application use cases.

- Split complex data binding into a command object. Make command objects rich [just like rich domain classes]. Creating a hierarchy of command objects can also be useful in some scenarios.

- Leverage filters to implement interceptors for capturing performance metrics (after view action) and validating security (before action).

- Use URLMappings Groovy file to define error-code response, including mapping to a custom, user-friendly error page.

**SERVICE**

- A service is the right candidate for complex business logic or coarse-grained code. If required, the service API can easily be exposed as a Restful/SOAP web service.

- Though transactional by default, services can be made non-transactional if none of their methods update the persistence store.

**DOMAIN**

- Use dynamic finders for querying domain objects with two fields and leverage the Grails console for prototyping tricky dynamic finder queries.

- If there are more than two fields, choose criteria queries; they come in handy for dynamic reporting and group queries.

- When using HQL, always use named params to avoid SQL injection risks.

- Use Command objects to handle complex and custom validations within a domain object. Put in common custom validations into shared validator file; that way it can be reused across the application.

- Always be sure to validate hasErrors (!) every time save () method is invoked on the domain object. Also, check its return value; otherwise, it would not throw any error and result in unexpected behaviors.

**COMMON**

- Grails is convention driven, so follow its convention religiously; doing so will aid in good code maintainability.

- With the exception of services, test as much as possible with unit tests rather than with integration tests. Unit tests are much faster to run/debug and also better enforce low coupling. Use mockDomain(), mockLogging () and so on. It is advisable to have code coverage of 70 percent and above.

- Use the Groovy console to test and explore code before putting it in closures. Doing so will avoid many runtime headaches.

- Develop reusable parts of the application as Grails plugins. These plugins can be tested individually and will remove complexity from the main application(s).

- Become familiar with the resources plugin for handling static resources.

- Understand the underlying technologies—namely Spring and Hibernate. The developer is likely to encounter issues modeling domain objects and getting them to work together. Grails is truly a high-productivity framework, but if the developer does not understand how Hibernate works, it is easy to get lost when trying to direct persistence behavior.

- Have an externalized property file to house properties that can be overridden; this provides the flexibility to change configurations on production without redeployment.
3.1.2 Design Patterns

In simple terms, design patterns are general reusable solutions to commonly occurring design problems. As such, they can speed up the development process by providing tested, proven development paradigms. Design patterns reside in the domain of modules and their interconnections.

Many Java design patterns are also applicable to Groovy, but there are some distinctions:

- Some patterns carry over directly (and can make use of normal Groovy syntax improvements for greater readability).
- Some patterns are no longer required because they are built right into the language or because Groovy supports a better way of achieving the intent of the pattern.
- Some patterns that have to be expressed at the design level in other languages can be implemented directly in Groovy, because of the way Groovy can blur the distinction between design and implementation.

As a framework, Grails is based on the MVC architectural design pattern. Further, the underlying components of Grails leverage many of the design patterns (for example, Site Mesh is based on the Decorator pattern).

The following design patterns are applicable to Groovy (refer to the links for more details):

- Abstract Factory Pattern
- Adapter Pattern
- Bouncer Pattern
- Chain of Responsibility Pattern
- Composite Pattern
- Decorator Pattern
- Delegation Pattern
- Flyweight Pattern
- Iterator Pattern
- Loan my Resource Pattern
- Null Object Pattern
3.1.3 Issues and Pitfalls

Despite the many factors that make Grails a great framework, it also comes with its own set of issues or “gotchas.” This section attempts to capture some of those encountered by the Sapient development team.

OVERALL ISSUES

- In cases where a domain object is stored in session scope, the object will become detached from the Hibernate session. That means the developer is unable to access any uninitialized, lazy-loaded relationships [user.role.privilege] that the object holds. The only way to fix this is to explicitly reattach the object to session.

- In Grails 1.0, if “object.discard()” was not invoked explicitly on an object that failed “object.validate()”, it resulted in having the object saved to the database.

- While redirecting in controllers, the param map would be lost if not passed explicitly to the redirected action—for example, Redirect [action:'actionname', params:param].

GORM ISSUES

- Despite trying its best to hide implementation details, Grails’ database access library, GORM, which is based on Hibernate ORM, still leaks abstractions on occasion.

- “Calling Save() is not persisting the data as expected” happens for two reasons:
  - Whenever a save() is invoked on any domain object, Grails validates against the constraint defined. If any of the constraints fails, it will attach the error to the domain silently. It will not explicitly raise it until the return value of save() or call hasErrors() is checked. It will then handle it accordingly by invoking “discard()”. The other way to handle this is by setting “failOnError” property on the domain object to true.

  ```groovy
def user = new User(params)
  if (!user.save()) {
    // Save failed! Present the errors to the user.
    User.discard();
  }
```

- Although there are no validation errors on the domain, at times the data may not get persisted immediately. This delay results from the way the Hibernate session treats Hibernate-managed objects. While the developer can explicitly save the object by flushing the session “flush:true” on the domain object, this tactic should be used with discretion.

  ```groovy
  User.save(flush:true)
  ```

- “Data getting persisted even without invoking Save()” results from a Hibernate concept called dirty checking—meaning it will check if any changes have occurred to the domain object after it is being retrieved from the session. If yes, it will persist the changes.

  ```groovy
def b = User.findByName(params.name)
b.name = b.name.reverse()
```

As illustrated in the code snippet above, whenever a get or findby query is fired by GORM, it will attach the object to a Hibernate session. Once it gets out of the closure, it will close it and do a dirty check, detecting any change in name property and persisting the data to database by default.

This “gotcha” can be avoided by using “read()” method provided by GORM. This method passes the dirty check, or in scenarios where query must be used, the developer will have to explicitly discard() the changes programmatically.

- “Exception: not-null property references a null or transient value:”—how many times have developers come across such an exception, when trying to save a “Many-to-One” domain object and wondered why?

  ```groovy
class Location {
  String city
}
```
class Author {
    String name
    Location location
}

def a = new Author(name: "Niall Ferguson",
location: new Location(city: "Boston"))
a.save()

A transient object is one not attached to the Hibernate session. In the above example, a new location object is being created. Thus, it is not part of the Hibernate session. For this work, it is necessary to save the location object first as below.

def l = new Location(city: "Boston")
l.save()

def a = new Author(name: "Niall Ferguson", location: l)
a.save()

This is not the right way to do it, however. Ideally, when saving the author, the location should get saved, as well. How can the developer achieve this? Here is where understanding cascading is very important. Cascading determines what type of actions, when applied to a domain instance, also apply to the relations of that instance.

This is where "belongsTo" comes into use. By modifying the domain class as below and then invoking "author.save", the previous error will disappear. That is because cascading is now enabled, and GORM will ensure that saves are cascaded from author to location.

class Location {
    String city
    static belongsTo = Author
}

An alternate way to define belongsTo is as below

class Location {
    String city
    static belongsTo = [author: Author]
}

The difference between this and the former is that this establishes a bidirectional relationship and allows definition of multiple cascading relationships.

- There is another very important aspect to understand when dealing with One-To-Many/Many-To-Many relationships in GORM: impedance mismatch behind the object-oriented languages and relational database. GORM uses collections to handle a one-to-many relationship internally.
  - When dealing with "One-To-Many," the object will not be saved to database until an explicit flush() has been fired. This is a symptom of discrepancy between the in-memory collection and the database data.
  - When dealing with parent-child relationships, be sure to establish cascading using the "belongsTo." Doing so ensures that deletes and saves are cascaded correctly.
  - Saves are always cascaded from the parent to its children. There is an exception, "org.springframework.dao.InvalidDataAccessApiUsageException: deleted object would be re-saved by cascade," which occurs when trying to delete specific children for a parent. Consider an author having many books and then deleting certain books, as follows:

    Def a = Author.get(1)
a.books*.delete()

Books are still in the author’s collection. When the session is flushed, it will try to recreate it because Hibernate’s dirty checking will kick in. Should the developer always explicitly clear the collection and explicitly delete the books? Deleting should be performed on a copy of the collection; otherwise, it will throw "concurrentModificationException."

- If you use belongsTo with a collection, explicitly set the cascade type to "all-delete-orphan" in the parent’s mapping block.
3.1.3 Issues and Pitfalls (continued)

- For unidirectional relationships, GORM creates a join table by default. When clearing the collection of books, the records are simply removed from that join table. Bidirectional relationships are mapped using a straight foreign key on the child table.

- Use the "addTo*[]" method when adding new domain objects as opposed to using a "new" construct. Using a "new" construct would result in "null pointe Exception" as addTo[] is null safe and implicitly creates the correct domain class.

- Use "removeFrom*[]" method while removing domain objects.

- What follows are important considerations when dealing with Many-To-Many relationships:

  - Deletes do not cascade; it is better to manage the deletes ourselves here rather than leaving it to the discretion of GORM.

  - While one side of the relationship must always have "belongsTo," it does not matter which side it has.

  - The "belongsTo" only affects the direction of cascading saves; it does not cause cascading deletes.

  - A join table will always be created while creating a "Many-To-Many" relationship.

- Error reporting has been an issue with versions of Grails prior to 2.x. Those versions spit out tons of lines of error stack trace on the console, which was difficult to digest and interpret. With Grails version 2.x and above, this has been improved drastically through a new errors view that analyzes stack traces and recursively displays problem areas in the code.

4. Benefits of using Grails

Grails uses the Groovy language. The intent of Groovy was not to replace Java but to coexist with it—offering developers a language that uses the best of both to facilitate rapid application development that Java otherwise failed to support.

In the dynamic and demanding market, the cost-effectiveness of an enterprise is determined by:

- Developer productivity
- Time to market for the application
- Ability to adapt to changes quickly
- Sustainable performance and scalability of the application

Groovy and Grails deliver against all of those criteria. By abstracting best-of-breed Java technologies and leveraging the Groovy language, Grails provides much-needed agility and productivity for faster time to market. Because it is based on the JVM framework, it also delivers scalability and sustainable performance.
4.1. Time to Market

With Grails, the developer’s efficiency increases tremendously, resulting in reduced development time and faster time to market.

**Developer Efficiency:** Grails’ and Groovy’s convention over configuration model shields the developer from noisy configuration. Compared to other traditional Java web frameworks, much of the boilerplate code can be set up in an application within an hour. Further, its rich features—namely plugins for all aspects of web development, templating and auto-wiring—drives developer efficiency by enabling greater focus on the real business problem and overall project quality.

**Excellent Java Integration:** Grails and Groovy provide a great ecosystem, where both Groovy and Java languages can coexist. As such, the developer can use all useful Java libraries along with Groovy—empowering the developer to make the best design choices on a case-by-case basis. This flexibility and interoperability help to reduce development cycle phases while saving a lot of time.

**Reusability:** The Grails framework reflects the DRY philosophy. It encourages reusing code as much as possible, thereby eliminating repetition. For the Sapient team, this made the code very modular and thus easy to maintain and refactor. It also resulted in far fewer hidden bugs and eliminated spaghetti code—translating into faster and easier future enhancements. In short, Groovy code is much more concise and expressive—a further advantage in advancing developer productivity.

**Agility:** Grails facilitates iterative development by helping teams “chunk out” good functionalities in short sprints, with regular customer checkpoints and feedback. With its great built-in support for unit and integration testing, Grails also encourages and supports test-driven development, which helps in maintaining good code quality and reducing defects in the application.

4.2. Cost Reduction

**Leaner Team:** By supporting much more efficient development, Grails and Groovy requires a much smaller team. Fewer resources help to reduce costs. Given the efficiency and precision of Groovy code, the application required fewer lines of code, which also meant fewer bugs and faster enhancements. Grails does not require a web container restart for small changes, which drove a considerable reduction in development time.

**Reduced Integration:** Grails comes with a good plugin ecosystem that provides the developer with various plugins (search, export, scheduler to Ajax widgets and more). The plugins are very easy to integrate into the existing framework, thereby decreasing integration time and supporting greater efficiency and reduced development cost.

**Low Maintenance:** Grails’ convention over configuration paradigm and Groovy language efficiency translated into less time spent on boilerplate code and XML configuration in addition to a smaller code set. It also enables any new developer to ramp up and understand the code very quickly and easily—a key factor in addressing maintenance complexity and cost goals.
5. WHEN TO USE GRAILS

A key question remains: "When does it make sense to use Grails?"

Since Grails and Groovy have been around for just over eight years, the framework and language are relatively new market entrants. Other Java frameworks, such as Struts, Wicket and Spring, have been around longer. Thus, many of the so-called Java shops have already adopted one of those frameworks and have made significant investments in developers, technologies and infrastructure. Meanwhile, startup companies are seeking the right technology to adopt to survive and be productive.

GRAILS FOR STARTUPS
Groovy as a language and Grails as a framework would be a good choice for startup organizations or companies that are looking for a technology stack that will enable them to churn out functionality with speed and quality and that are leaning toward being a Java shop. As mentioned previously, a company’s survival depends on its ability to accelerate time to market and optimize costs. Grails and Groovy can deliver against those goals.

GROOVY FOR ENTERPRISE JAVA SHOPS
For organizations and companies that are already “Java Shops”—and that have made heavy investments in Java—adopting Grails should be fairly smooth and easy. The framework is a natural fit in an enterprise setting for several key reasons.

First, given Groovy’s flat learning curve and backward compatibility with Java, enterprise Java developers will easily transition to the dynamic language. Because Groovy is the JVM language and is based on Java, a developer will not have to relearn the basics. Most enterprises have a multitude of native and third-party Java products. With Grails Java support, it becomes easier for the developer to integrate with them.

Second, most enterprises have already made an investment in Spring and Hibernate. Since the Grails framework is built on top of Hibernate and Spring, it is far easier not only to understand the framework but also to address any issues that arise.

Finally, when we talk about an enterprise other than the technology stacks, much investment in middleware/support has already been made following application deployments, application servers, monitoring, clustering, HA and load balancing. Given that Grails uses Spring and Hibernate “under the hood” and that Groovy translates to byte code, deploying a Grails app is not different from any J2EE-based application. Hence, no additional middleware or support investment is required; the existing infrastructure can be leveraged for deploying, running and maintaining the Grails application.

GROOVY WITH EXISTING DATABASES
An enterprise typically has a well-defined database and/or domain model, with multiple applications interacting with it. In most cases, applications are built using these existing database tables (with more tables added as needed). Thus, Java enterprises likely have existing domain models based on Hibernate or JPA.

Fortunately, Grails can still use these existing models. Include the XML Hibernate mapping or JPA Annotations in the Grails project as Java source or as a JAR, create a Hibernate configuration file and then add GORM constraints in separate files. Doing so provides full access to all of Grails’ great features, such as dynamic finders and Where queries. Refer here for complete details on how to do this.
What can be done if an enterprise lacks an existing domain model and would prefer to have a full-blown standard GORM model? In that case, Grails provides the Reverse Engineering tool, which helps generate the GORM model for an existing database schema. Since it is not out of the box, the model will require some tweaking. Even so, it saves a lot of time if there are many database tables.

From the past experience in an enterprise context, the right approach would be to leverage multiple ORM frameworks in an application. Doing so keeps things simpler, more modular and less complicated. GORM could be used for all the domains for which a CRUD operation is essential; for only Selects, IBATIS—a lightweight data mapping framework and persistence API—can be leveraged. IBATIS also provides good performance gains by enabling the developer to specify various options for database connection pooling, result caching and transaction management. It is also easy to integrate and use in conjunction with GORM within the Grails framework.

As Matt Raible says, when it comes to web frameworks, “There is no best framework, just a lot of awesome choices.” Comparing web frameworks is still not an established discipline, and neutrality has not yet been reached. Nevertheless, Raible, who has been presenting Java web framework comparisons since 2007, came up with a comparison of frameworks (below). Though opinionated, his assessment is quite interesting.

As part of the comparison, he devised a matrix comparing each framework against various criteria. He also provides weighted ratings and uses the total weightage to determine the top three frameworks. The numbers in the matrix and the criteria for choosing them are just Raible’s opinion. In short, a framework earns a 0 if it does not have a characteristic at all, 0.5 if it has it at some level, and 1 if it is fully satisfied.

Based on Raible’s assessment, the Grails, spring, Rails and GWT emerge as “winners.”

### 6. COMPARISON AND ADOPTION

![Comparison Matrix](image)
In a community research article entitled “Top 20 Web Frameworks for the JVM,” InfoQ asked the community to rate each of the 20 frameworks identified based on relative importance and maturity.

Adoption Readiness was broken into four categories: Hype, Assess, Trial and Adopt. Each participant was asked to select either of these categories when voting. What follows are the rankings based on the votes received from the community.

In this assessment, Spring MVC, Play and Grails are the top three Java web frameworks. The heatmap for Spring MVC and Grails (below) shows greater concentration on Adoption Readiness and relative Importance, suggesting that these are relatively mature frameworks in terms of enterprise adoption.

Source: InfoQ – “Top 20 Web Frameworks for JVM”
Since its inception in 2003, the Grails and Groovy open-source project has been very active. Now under VMware, the framework and language are supported by a dedicated team. VMware has already shared a detailed roadmap, which outlines carefully planned future milestones. (The current production versions are Grails 2.2 and Groovy 2.0. VMware has Grails 2.2.x slated for early 2013, and Groovy 3.0 slated for 2014.) Additionally, VMware offers solid documentation and a vibrant Grails community, which helps in obtaining quick resolution to any problems that arise.

KEY GROOVY 2.0 FEATURES

- **Static type checking:** This feature brings compilation checks into the otherwise runtime checks that occur with the Groovy dynamic language. The static type checker is built using Groovy’s existing powerful Abstract Syntax Tree (AST) transformation mechanisms. For those not familiar with these mechanisms, think of them as an optional compiler plugin triggered through an annotation. Because static type checking is an optional feature, the developer is not forced to use it. To trigger static type checking, simply use the @TypeChecked annotation on a method or on a class to turn on checking at your desired level of granularity. This feature renders Groovy faster as Java.

- **More performance:** For those fortunate to have JDK 7 already in production, the support of JDK 7 Invoke Dynamic speeds up Groovy. It also offers static compilation for JDK 5 and beyond for everyone, and particularly those ready to abandon some aspects of dynamicity to shield themselves from the reach of “monkey patching” and to gain the same speed as Java.

- **More Java friendliness:** When it comes to syntax, the Java 7 Project Coin enhancements keep Groovy and Java as close cousins as ever. For developers using Groovy as a Java scripting language, the static type checker provides the same level of feedback and type safety as the javac compiler.

- **More modularity:** With a new level of modularity, Groovy opens the doors for smaller deliverables (for example, integration in mobile applications on Android). It also allows the Groovy APIs to grow and evolve with newer versions and newer extension modules while also allowing users to contribute extension methods to existing types.

KEY GRAILS 2.2 ENHANCEMENTS

- **Interactive mode and console enhancements:** Grails 2.0 features brand new console output that is more concise and user friendly to consume. In general, Grails makes its best effort to display update information on a single line and present only crucial information. In previous versions of Grails, the war command produced many lines of output; in Grails 2.0, only one line of output is produced.

- **Reloading agent:** Grails 2.0’s reloading mechanism no longer uses class loaders, relying instead on a JVM agent to reload changes to class files. In addition to driving greatly improved reliability when reloading changes, it also ensures that the class files stored in disk remain consistent with the class files loaded in memory. That, in turn, reduces the need to run the clean command.

- **Performance improvements and HTML5 scaffolding:** Performance of GSP page rendering has once again been improved by optimizing the GSP compiler to inline method calls where possible.

- **Multiple data sources with Hibernate 3.6 support:** It is now possible to define multiple data sources in DataSource.groovy and to declare one or more data sources a particular domain uses by default. It also has new database migration and reverse engineering plugin support.

- **Integration with Maven and Gradle Build Tool:** Versions of Grails 2.1 and 2.2 will integrate support for Maven and Gradle build tools, respectively.

Spring 4.0 milestone releases are tentatively slated for 2014, with plans to include support for Groovy and Java SE 8—indicating that Grails and Groovy are here to stay.
8. CONCLUSION

As a dynamic web framework embracing DRY principles, Grails was designed to be a companion to, rather than a replacement for, Java. Grails can dramatically reduce the complexity of building web applications, helping Java developers create applications with greater speed, agility and flexibility in terms of integrating with existing systems. As such, it can help in safeguarding existing Java investments.

Grails is not alone in the realm of dynamic Java web frameworks. Other frameworks, such as Play and Lift, were inspired by peer frameworks, such as Ruby on Rails. Some of these frameworks share convention over configuration philosophy but leverage Scala as their programming language. Like Grails, these frameworks are gaining in popularity, illustrating a growing trend in rapid application development Java frameworks.

Using a real-world scenario, this paper aimed to provide a holistic review of Grails as a Java web framework. In discussing how Grails can be adopted and integrated into an enterprise, this paper merely scratches the surface. Like other Java web frameworks, Grails and Groovy present their own sets of issues and challenges, as detailed in this paper. However, when we weigh the pros against the cons, Grails and Groovy remain a great framework and language.

9. REFERENCES

http://grails.org/doc/latest/


http://groovy.codehaus.org/Design+Patterns+with+Groovy

http://jaxenter.com/is-grails-suitable-for-the-java-enterprise-you-bet-43535.html

http://www.infoq.com/articles/grails-best-practices


http://blog.springframework.org/2010/07/02/gorm-gotchas-part-2/

http://blog.websitesframeworks.com/2013/03/web-framework-comparison-matt-raible-opinion-138/

http://java.dzone.com/articles/infoqs-top-20-web-frameworks

http://www.infoq.com/research/jvm-web-frameworks

Source: Grails in Action

Source: Spring Source
<table>
<thead>
<tr>
<th>City</th>
<th>Address</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bangalore</td>
<td>Salarpuria GR Tech Park 6th Floor, &quot;VAYU&quot; Block #137, Bengaluru 560066</td>
<td>Tel: +91 [080] 410 47 000</td>
</tr>
<tr>
<td>Geneva</td>
<td>Succursale Genève c/o Florence Thiébaud, avocate rue du Cendrier 15</td>
<td>Tel: +41 [0] 58 206 06 00</td>
</tr>
<tr>
<td>Houston</td>
<td>Heritage Plaza 1111 Bagby Street Suite 1950</td>
<td>Tel: +1 [713] 493 6880</td>
</tr>
<tr>
<td>London</td>
<td>Eden House 8 Spital Square London, E1 6DU</td>
<td>Tel: +44 [0] 207 786 4500</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>1601 Cloverfield Blvd. Suite 400 South</td>
<td>Tel: +39-02-36269529</td>
</tr>
<tr>
<td>Delhi</td>
<td>Unitech Infospace Ground Floor, Tower A Building 2, Sector 21 Old Delhi - Gurgaon Road Dundahera, Gurgaon 122016 Haryana India Tel: +91 [124] 499 6000</td>
<td></td>
</tr>
<tr>
<td>Milan</td>
<td>Sapient Italy S.r.l Viale Bianca Maria 23 20122 Milan Italy</td>
<td>Tel: +39-02-36269529</td>
</tr>
<tr>
<td>Mumbai</td>
<td>Sapient Consulting Pvt. Ltd R-Tech Park, Goregaon(E) 13th Floor, Building 2, Off Western Express Highway Mumbai, Maharashtra - 400063 India Tel: +91-22-44764567</td>
<td></td>
</tr>
<tr>
<td>Noida (NCR of Delhi)</td>
<td>&quot;Oxygen&quot;, Tower C, Ground - 3rd floor Plot No. 7, Sector 144 Expressway Noida 201304 Uttar Pradesh India Tel: +91 [120] 479 5000</td>
<td></td>
</tr>
<tr>
<td>New York</td>
<td>40 Fulton Street 22nd Floor</td>
<td>Tel: +1 [212] 206 1005</td>
</tr>
<tr>
<td>Toronto</td>
<td>129 Spadina Avenue Suite 500</td>
<td>Tel: +65 6671 4933</td>
</tr>
<tr>
<td>Singapore</td>
<td>158 Cecil Street, #03-01</td>
<td>Tel: +65 6671 4933</td>
</tr>
<tr>
<td>Washington DC</td>
<td>1515 North Courthouse Road 4th Floor</td>
<td>Tel: +1 (703) 908 2400</td>
</tr>
<tr>
<td>Zürich</td>
<td>Seefeldstrasse 35 8008 Zürich Switzerland</td>
<td>Tel: +41 (58) 206 06 00</td>
</tr>
<tr>
<td>Frankfurt</td>
<td>Skyper Villa Taunusanlage 1 60329 Frankfurt Germany</td>
<td>Tel: +49 [0]69 505060594</td>
</tr>
<tr>
<td>Munich</td>
<td>Arnulfstrasse 60 80335 München Germany</td>
<td>Tel: +49 (0) 89 552 987 0</td>
</tr>
</tbody>
</table>