

Введение в Grails

Андрей Уваров

Разработка веб-приложений никогда не была занятием очень простым и быстрым. И множество разработчиков по всему миру стараются как-то эту ситуацию исправить. Но в то время как одни люди ищут, другие уже нашли свой путь. «Нам не нужна серебряная пуля, когда мы уже нашли Священный Грааль», – говорят разработчики Grails.

Grails (<http://grails.org>) – это фреймворк, предназначенный для разработки веб-приложений на языке Groovy. Данный фреймворк был написан Java-разработчиками для Java-разработчиков. GRAILS появился летом 2005 года, когда Graeme Keith Rocher, Steven Devijver и Guillaume Laforge поняли, что устали от существующего процесса разработки веб-приложений, и решили эту ситуацию как-то исправить. Как видно уже из названия, большое влияние на него оказал Ruby On Rails, и потому у данных фреймворков существует очень много схожих черт. Тем, кто уже имеет опыт работы с Ruby On Rails или Pylons, будет довольно просто разобратся в Grails.

Groovy (<http://groovy.codehaus.org>) – объектно-ориентированный язык программирования с возможностью динамической типизации. Разработан для платформы JVM (Java Virtual Machine). Таким образом в Groovy можно использовать любой Java-код и наоборот. Авторы языка постарались сделать его максимально понятным для Java-разработчиков, но при этом наделить возможностями, которых нет в Java, как, например, динамической типизацией, перегруз-

кой операторов и встроенным синтаксисом для работы с массивами, списками и т. п.

Обычный подход

Попробуем разобраться, из-за чего могло возникнуть желание придумать новый подход к процессу разработки веб-приложений.

Обычно веб-разработка на Java – процесс весьма утомительный, а говорить о скорости вообще не приходится. Вот только представьте себе, сколько действий необходимо сделать, чтобы добавить в приложение новую страницу:

- Создать шаблон страницы. Шаблон в данном случае есть представление.
- Создать DTO-класс (Data Transfer Object), на самом деле это класс модели для MVC.
- Создать класс Assembler, который будет уметь формировать из доменных объектов наш DTO-объект.
- Создать бизнес-фасад и его имплементацию (хотя это не всегда необходимо). Бизнес-фасады являются объектами, выполняющими сложную логику приложения, зачастую методы бизнес-фасадов выполняются транзакционно.

- Создать DAO-интерфейс (Data Access Object) и его имплементацию. DAO-объекты используются для работы с базой данных.
- Создать класс-контроллер, который будет обрабатывать поступающие запросы, вызывать нужные методы нашего фасада и собирать вместе модель и отображение (имеются в виду составляющие части MVC) и возвращать результат.
- И все созданные классы сконфигурировать при помощи Spring.

На самом деле данный процесс описывает общий случай, и далеко не всегда нам нужны, например, DTO-или Assembler-классы. Самым утомительным обычно является процесс конфигурации. Так как приходится создавать много XML-кода, описывающего и инициализирующего наши объекты, а также связи между ними.

Как это делается в Grails

Приведённая дальше схема является не единственной, и, может, программистам, не знакомым с Java и Spring, будет не очень понятно, как это всё работает. Но самое главное, что видно, так это сложность и длительность процесса. В противовес этому процесс

создания новой страницы в существующем Grails-приложении следующий:

- Создать новый шаблон страницы.
- Создать новый контроллер.
- Создать новый сервис. В рамках Grails сервис – есть класс, который должен выполнять сложную бизнес-логику, и методы сервиса могут быть транзакционными.

Последний шаг не всегда необходим. И самое важное – не требуется никакого XML!

Гораздо проще, не так ли?

Ингредиенты

Итак, прежде чем приступить к примерам и подробностям, необходимо разобратся с базовыми вещами.

В основе Grails лежат:

- **Spring** (<http://springframework.org>) – используется для реализации MVC-паттерна, и также может быть применён для использования уже существующего Java-кода в новом приложении.
- **Hibernate** (<http://www.hibernate.org>) – наиболее популярный и функциональный ORM-фрэймворк (Object Relationship Mapping). Грубо говоря, благодаря Hibernate мы можем обращаться с таблицами в базе, как с объектами. Например, если у нас есть в базе таблица 'user', содержащая информацию о пользователях, то при помощи Hibernate мы можем выбирать записи из этой таблицы как экземпляры класса User, а не просто массивы данных.
- **Sitemesh** (<http://www.opensymphony.com/sitemesh>) – специальный движок для работы с шаблонами страниц.

В Grails существует некоторое количество понятий, с которыми стоит ознакомиться первым делом:

- Доменные объекты.
- Контроллеры.
- Сервисы.
- Библиотеки тегов.
- Шаблоны.
- Задачи.

Начнём по порядку. Доменные объекты являются кирпичиками веб-приложения, понятиями предметной области. Так, например, если мы пишем электронный магазин, то у нас, скорее

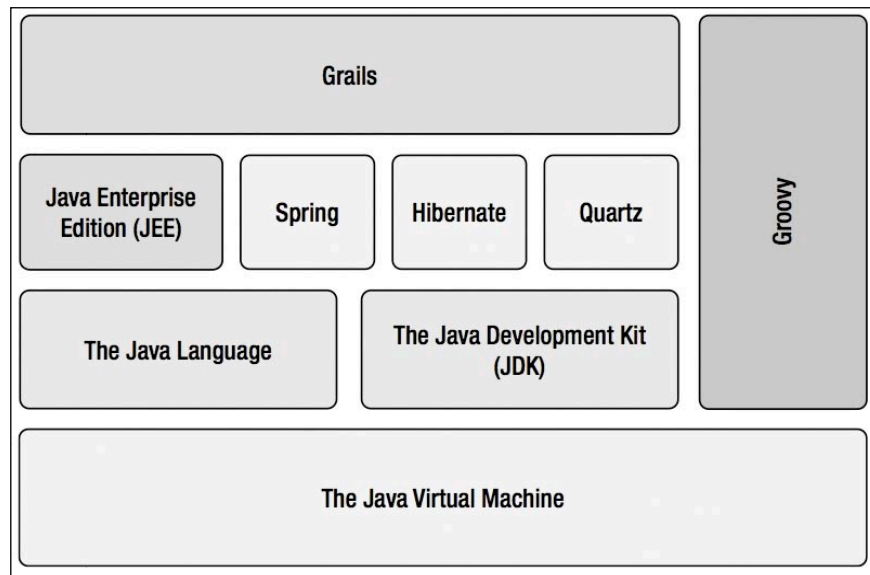


Рисунок 1. Архитектура Grails

всего, будут такие объекты, как «Пользователь», «Заказ», «Товар» и прочие. По доменным объектам Grails автоматически генерирует таблицы в базе данных вместе с ключами, ограничениями, индексами и всем-всем. Запросы к базе также в основном осуществляются посредством этих объектов, наподобие использования ActiveRecord в Ruby On Rails.

Когда пользователь набирает URL нашего сайта, то первым делом запрос попадает в контроллер этого запроса. Контроллер получает пользовательские данные, генерирует на их основе модель и передаёт её в отображение.

Сервисы – специальные сущности, задачей которых является выполнение бизнес-логики приложения. Как уже было сказано выше – в Java-приложениях для выполнения бизнес-логики используются бизнес-фасады.

Шаблоны, в общем-то как и везде, являются HTML-клише, по которым генерируются страницы. В Grails с целью упрощения создания шаблонов существует встроенный язык – GSP (Groovy Server Pages), он имеет теги, позволяющие осуществлять разнообразные действия, такие как итерация, условия, вывод данных и прочие. Но вы можете сами расширять набор тегов, создавая свои собственные библиотеки тегов.

Библиотекой тегов в Grails является класс, методы которого могут быть вызваны из шаблона как GSP-теги. Если в шаблонах часто приходится выполнять одинаковые действия, тогда сто-

ит создать специальный тег и перенести эти действия в него. Таким образом вы избежите рутинной работы и будете повторно использовать существующий код, а также облегчите жизнь верстальщику, работающему с шаблонами.

Уже ранее упоминалось, что Spring является одной из составляющих Grails. Spring в своём составе имеет специальную функциональность (Quartz) для выполнения задач по расписанию, то есть делать какую-то необходимую нам работу через определённые интервалы времени (разумеется можно выполнять задачи и в чётко фиксированное время). Создавая задачи таким образом, вы гораздо в меньшей степени зависите от операционной системы, чем, например, в случае с использованием Cron.

Создание приложения

Теперь можно перейти к практике. Создадим простое приложение – библиотеку. Наша библиотека будет просто хранить список книг и будет предоставлять возможность просматривать список, удалять, редактировать и добавлять книги в него.

Первым шагом установим непосредственно фрэймворк. Данный процесс представляется весьма простым – необходимо скачать архив с официального сайта (<http://grails.org>), распаковать и добавить в переменные окружения новую переменную GRAILS_HOME, которая будет указывать путь к grails. И затем добавить в переменную PATH

значение `#{GRAILS_HOME}/bin`. Подразумевается, что Java и Groovy уже установлены, в любом случае их установка не представляет какой-либо сложности.

Для выполнения рутинных задач – создания скелета простого приложения, доменных объектов, контроллеров и прочего – в комплект входит командная утилита `grails`. Подробно на командах останавливаться не будем, но самые важные рассмотрим в ходе написания нашего приложения. И первая команда, с которой мы познакомимся, – `create-app`.

```
bash-3.2$ grails create-app homeLibrary
```

```
Welcome to Grails 1.0.1 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: /Library/grails-1.0.1

Base Directory: /demo/grails
Environment set to development
Note: No plugin scripts found
Running script /Library/grails-1.0.1/scripts/CreateApp.groovy
[mkdir] Created dir: /demo/grails/homeLibrary/src
[mkdir] Created dir: /demo/grails/homeLibrary/src/java
[mkdir] Created dir: /demo/grails/homeLibrary/src/groovy
[mkdir] Created dir: /demo/grails/homeLibrary/grails-app
[mkdir] Created dir: /demo/grails/homeLibrary/grails-app/controllers
[mkdir] Created dir: /demo/grails/homeLibrary/grails-app/services
[mkdir] Created dir: /demo/grails/homeLibrary/grails-app/domain
[mkdir] Created dir: /demo/grails/homeLibrary/grails-app/taglib
[mkdir] Created dir: /demo/grails/homeLibrary/grails-app/units
[mkdir] Created dir: /demo/grails/homeLibrary/grails-app/views
[mkdir] Created dir: /demo/grails/homeLibrary/grails-app/views/layouts
[mkdir] Created dir: /demo/grails/homeLibrary/grails-app/i18n
[mkdir] Created dir: /demo/grails/homeLibrary/grails-app/conf
[mkdir] Created dir: /demo/grails/homeLibrary/test
[mkdir] Created dir: /demo/grails/homeLibrary/test/unit
[mkdir] Created dir: /demo/grails/homeLibrary/test/integration
[mkdir] Created dir: /demo/grails/homeLibrary/scripts
[mkdir] Created dir: /demo/grails/homeLibrary/web-app
[mkdir] Created dir: /demo/grails/homeLibrary/web-app/js
[mkdir] Created dir: /demo/grails/homeLibrary/web-app/css
[mkdir] Created dir: /demo/grails/homeLibrary/web-app/images
[mkdir] Created dir: /demo/grails/homeLibrary/web-app/META-INF
[mkdir] Created dir: /demo/grails/homeLibrary/lib
[mkdir] Created dir: /demo/grails/homeLibrary/grails-app/conf/spring
[mkdir] Created dir: /demo/grails/homeLibrary/grails-app/conf/hibernate
[propertyfile] Creating new property file: /demo/grails/homeLibrary/application.properties
[copy] Copying 2 files to /demo/grails/homeLibrary
[copy] Copying 2 files to /demo/grails/homeLibrary/web-app/WEB-INF
[copy] Copying 5 files to /demo/grails/homeLibrary/web-app/WEB-INF/tld
[copy] Copying 87 files to /demo/grails/homeLibrary/web-app
[copy] Copying 17 files to /demo/grails/homeLibrary/grails-app
[copy] Copying 1 file to /demo/grails/homeLibrary
[copy] Copying 1 file to /demo/grails/homeLibrary
[copy] Copying 1 file to /demo/grails/homeLibrary
[propertyfile] Updating property file: /demo/grails/homeLibrary/application.properties
Created Grails Application at /demo/grails/homeLibrary
```

Выполнив команду «`grails create-app homeLibrary`», мы имеем скелет нашего будущего веб-приложения. В Grails имеет место такое понятие, как *Convention over configuration*. Это означает, что для каждого файла существует своё место, где он должен находиться, и каждый класс должен иметь чётко регламентированное название (см. **рис. 2**). Например, файлы, содержащие классы контроллеров, должны иметь постфикс `Controller`.

Конфигурирование базы данных

В качестве следующего этапа настроим приложение для работы с базой данных, пусть это будет MySQL, но подойдёт и любая другая СУБД, с которой умеет работать Hibernate (на данный момент существует поддержка всех популярных СУБД). Для этого откроем файл `conf/DataSource.groovy`:

```
dataSource {
    pooled = false
    // Используемый нами драйвер
    driverClassName = "com.mysql.jdbc.Driver"
    username = "root"
    password = "root"
}
hibernate {
    cache.use_second_level_cache=true
```

```
cache.use_query_cache=true
cache.provider_class='org.hibernate.cache.EhCacheProvider'
}
// environment specific settings
environments {
    development {
        dataSource {
            dbCreate = "create-drop"
            // one of 'create', 'create-drop', 'update'
            // Строка подключения к базе данных.
            // Предполагается, что база home_library
            // уже создана
            url = "jdbc:mysql://127.0.0.1:3306/home_library"
        }
    }
    test {
        dataSource {
            dbCreate = "update"
            url = "jdbc:hsqldb:mem:testDb"
        }
    }
    production {
        dataSource {
            dbCreate = "update"
            url = "jdbc:hsqldb:file:prodDb;shutdown=true"
        }
    }
}
```

Как видим, данный файл состоит из трёх секций: `dataSource`, `hibernate` и `environments`. Секция `dataSource` предназначена для настройки параметров доступа непосредственно к СУБД. Так как мы используем MySQL, то соответственно изменим имя драйвера на «`com.mysql.jdbc.Driver`», а сам архив с драйвером необходимо поместить в директорию «`lib`» нашего проекта. Данная директория предназначена для хранения используемых Java-библиотек. Следующая секция предоставляет нам возможность настройки Hibernate и на данном этапе неинтересна, посему перейдём сразу к конфигурированию окружений – `environments`.

Grails-приложение может выполняться в трёх различных окружениях (режимах): режим разработки, тестирования и готового продукта. Для каждого отдельного режима мы можем задать URL, логин, пароль для подключения к СУБД и прочее. По умолчанию приложение запускается в режиме разработки. Если существует необходимость, то можно самому создать дополнительные режимы. Так как мы уже выбрали в качестве СУБД MySQL, то определим строку подключения: `jdbc:mysql://127.0.0.1:3306/home_library`.

Доменные объекты

Как уже упоминалось, доменные объекты являются кирпичиками приложения, его базовыми понятиями. Так как наше приложение призвано хранить список книг, то создадим доменный объект `Book`.

```
bash-3.2$ grails create-domain-class Book
```

```
Welcome to Grails 1.0.1 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: /Library/grails-1.0.1

Base Directory: /demo/grails/homeLibrary
Environment set to development
Note: No plugin scripts found
Running script /Library/grails-1.0.1/scripts/CreateDomainClass.groovy
[copy] Copying 1 file to /demo/grails/homeLibrary/grails-app/domain
Created for Book
[copy] Copying 1 file to /demo/grails/homeLibrary/test/integration
Created Tests for Book
```

Как видите, с этим нам вновь помогла команда `grails`. Ввиду того, что каждый экземпляр класса `Book` будет пред-

ставлять собой информацию о конкретной книге, соответственно добавим необходимые поля:

```
class Book {
    String name // Название книги
    String author // Имя автора
    int shelfNumber // Номер полки
    static constraints = {
        // Зададим ограничения
        name(nullable: false, blank: false, size: 1..200)
        author(nullable: false, blank: false, size: 1..50)
    }
}
```

Теперь при запуске приложение на основе нашего доменного объекта Grails создаст в базе таблицу «book». Такой метод создания доменных объектов и связывания их с базой представляется несколько менее трудоёмким, чем традиционно используемые в Java аннотации или файлы маппингов. Чтобы убедиться, выполним команду «grails run-app» и посмотрим, что изменилось в базе.

```
mysql> show create table book\G
```

```
***** 1. row *****
Table: book
Create Table: CREATE TABLE `book` (
  `id` bigint(20) NOT NULL auto_increment,
  `version` bigint(20) NOT NULL,
  `author` varchar(50) NOT NULL,
  `name` varchar(200) NOT NULL,
  `shelf_number` int(11) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1
1 row in set (0.00 sec)
```

Как видим, в таблице присутствуют все определённые нами поля и ограничения. Также автоматически был добавлен первичный ключ и служебное поле «version». Так как для окружения разработки в конфигурационном файле DataSource.groovy опция «dbCreate» имеет значение «create-drop», то при каждом запуске приложения таблицы будут удаляться и создаваться вновь. Разумеется, все данные будут при этом утеряны. Чтобы этого избежать, необходимо заменить значение на «update». В этом случае, если будут происходить какие-либо изменения наших доменных объектов, то на существующих таблицах будут выполняться SQL-запросы ALTER, и, соответственно, потери данных будут иметь место только в случае удаления каких-либо полей или классов.

Контроллер и шаблоны

Теперь, когда у нас есть уже объект предметной области – Book, мы можем создать интерфейс для управления книгами. Тут существует несколько путей – мы можем написать всё вручную, можем использовать так называемый scaffolding, а можем сгенерировать по существующему доменному классу всё необходимое, а именно –



Рисунок 3. Главная страница

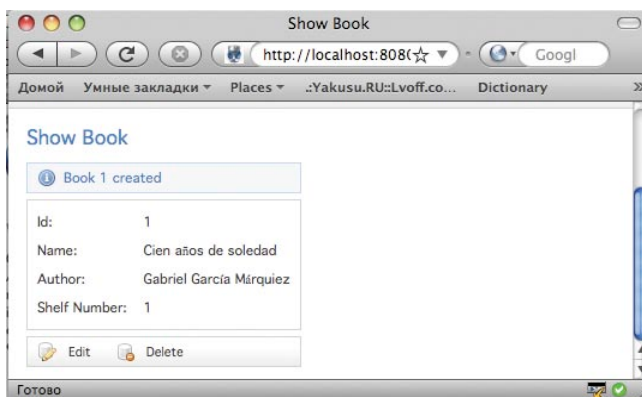


Рисунок 4. Простое CRUD-приложение (Create Retrieve Update Delete)

шаблоны и контроллер (scaffolding – поведение контроллера, когда явно не существует никаких методов и шаблонов, но при этом контроллер может создавать, удалять и редактировать данные. Например, мы можем создать пустой контроллер, с полем «def scaffold = Book», и таким образом теперь получим возможность выполнять через данный контроллер простые действия с объектом Book). Чтобы продемонстрировать возможности Grails, попробуем пойти по третьему пути и сгенерировать код и шаблоны. Вновь на помощь нам приходит утилита grails. Выполним команду «grails generate-all Book» и запустим приложение.

Открыв браузер, мы увидим главную страницу с описанием и списком из одной ссылки – ссылки на сгенерированный только что контроллер (см. рис. 3).

Перейдя по ней, убедимся, что наше приложение действительно может создавать новые записи, редактировать и удалять (см. рис. 4).

Чтобы при обращении к нашему приложению сразу можно было приступить к работе с книгами, отредактируем конфигурационный файл conf/UrlMappings.groovy:

```
class UrlMappings {
    static mappings = {
        "/$controller/$action?/$id?" {
            constraints {
                // apply constraints here
            }
        }
        "500" (view: '/error')
        // Переопределим поведение по умолчанию
        "/" (controller: 'book')
    }
}
```

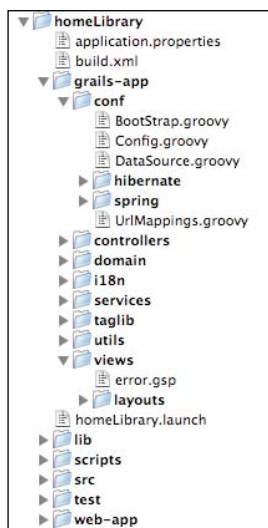


Рисунок 2. Структура приложения

Читатели, знакомые с Ruby On Rails, заметят сходство с routes. Первая часть адреса (после имени домена) показывает, какой контроллер необходимо использовать для обработки запроса, следующая часть определяет метод, которым надо выполнять обработку, и последняя часть является значением ID. ID представляет собой всего лишь GET-параметр с именем «ID».

Чтобы лучше понимать работу приложения, проследуем по всем этапам, от запроса в браузере до формирования ответа. Итак, теперь, если вы обратитесь по адресу: `http://localhost:8080/homeLibrary`, то приложение, согласно правилам, определённым в `UrlMappings.groovy`, передаст управление в контроллер `BookController.groovy`:

```
class BookController {

    def index = { redirect(action:list,params:params) }
    // the delete, save and update actions only accept
    // POST requests
    def allowedMethods = [delete:'POST', save:'POST',
        update:'POST']

    def list = {
        if(!params.max) params.max = 10
        [ bookList: Book.list( params ) ]
    }

    def show = {
        def book = Book.get( params.id )
        if(!book) {
            flash.message = "Book not found with id
                id ${params.id}"
            redirect(action:list)
        }
        else { return [ book : book ] }
    }

    def delete = {
        def book = Book.get( params.id )
        if(book) {
            book.delete()
            flash.message = "Book ${params.id} deleted"
            redirect(action:list)
        }
        else {
            flash.message = "Book not found with id
                id ${params.id}"
            redirect(action:list)
        }
    }

    def edit = {
        def book = Book.get( params.id )
        if(!book) {
            flash.message = "Book not found with id
                id ${params.id}"
            redirect(action:list)
        }
        else {
            return [ book : book ]
        }
    }

    def update = {
        def book = Book.get( params.id )
        if(book) {
            book.properties = params
            if(!book.hasErrors() && book.save()) {
                flash.message = "Book ${params.id} updated"
                redirect(action:show,id:book.id)
            }
            else {
                render(view:'edit',model:[book:book])
            }
        }
        else {
            flash.message = "Book not found with id
                id ${params.id}"
            redirect(action:edit,id:params.id)
        }
    }
}
```

```
def create = {
    def book = new Book()
    book.properties = params
    return ['book':book]
}

def save = {
    def book = new Book(params)
    if(!book.hasErrors() && book.save()) {
        flash.message = "Book ${book.id} created"
        redirect(action:show,id:book.id)
    }
    else {
        render(view:'create',model:[book:book])
    }
}
```

Так как явно не указан метод, который необходимо вызвать, то по умолчанию вызывается метод – «index» (при желании можно переопределить метод, по умолчанию вызывающийся следующим образом – добавить в контроллер строку: «`def defaultAction = 'show'`»). Как видно из приведённого кода, метод «index» просто перенаправляет пользователя при помощи HTTP-заголовков на другой метод этого же класса – «list». Все POST- и GET-параметры доступны через поле `params`, которое представляет собой ассоциативный массив.

В методе «list» извлекается из базы список всех существующих книг и возвращается в качестве модели. Затем, незаметно для нас, Grails находит в директории «`views/book/`» шаблон «`list.gsp`» и передаёт внутрь только что сгенерированную модель. Как видите, и здесь работает правило «Convention Over Configuration», когда фреймворк сам знает, где найти необходимые шаблоны. После того как по нашему шаблону и модели сгенерирован HTML-код, к работе приступает Sitemesh. В нашем шаблоне был использован специальный GSP-тег «`<meta name="layout" content="main" />`». Данный тег показывает, что для финального отображения страницы необходимо использовать лейаут «main» (все лейауты находятся в директории «`views/layouts`»). И теперь Sitemesh применяет к уже обработанному шаблону страницы указанный лейаут и выводит результат.

Заключение

Как видите, всё весьма просто и прозрачно, чего, собственно, и добивались разработчики. К сожалению, рассмотреть все возможности Grails в одной статье не представляется возможным, посему такие вещи, как сервисы, библиотеки тегов, выполнение задач по расписанию, взаимодействие с Java-кодом, тонкости работы с базой и многое другое, не были рассмотрены.

В завершении хотелось бы оценить преимущества и недостатки данного фреймворка. Используя Grails, построить большое веб-приложение Enterprise-уровня со сложной бизнес-логикой так же сложно, как и обычными средствами Java. Это основной недостаток. Но для решения большинства повседневных задач имеется полный набор инструментов. К преимуществам, несомненно, стоит отнести простоту разработки, возможность запуска полученных приложений на любых Java-веб-серверах (то есть контейнерах или серверах приложений) и возможность использования уже существующего Java-кода с минимальными затратами на его адаптацию. 