

Developing Java enterprise applications in minutes with Spring Roo

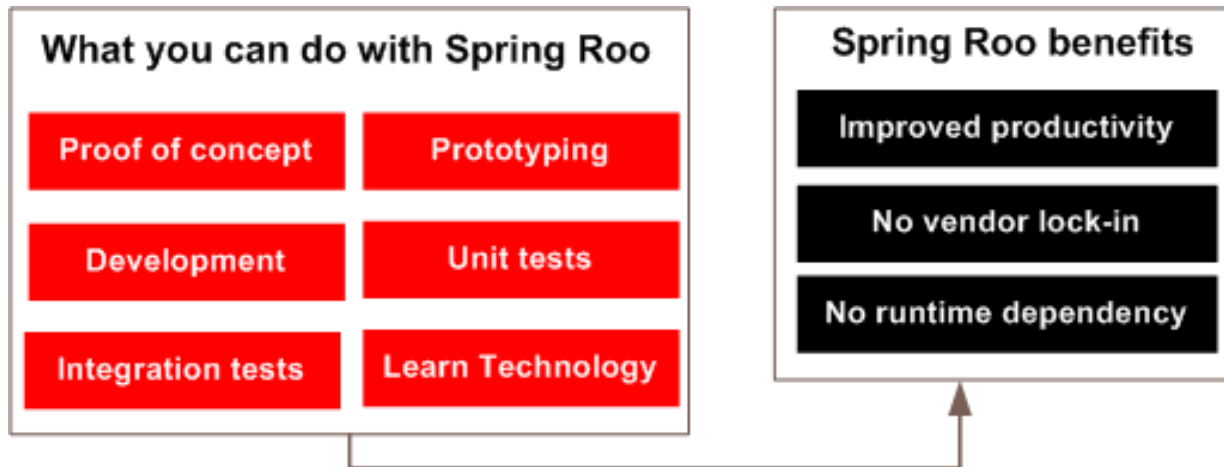
Ashish Sarin

Agenda

- The **What** and the **Why** of Spring Roo
- Spring Roo **CLI**
- Demo – Flight Booking application
- Customizing Roo-generated configuration and code
- What more you can do with Spring Roo
- Spring Roo architecture
- Extending Roo with custom add-ons
- The road ahead

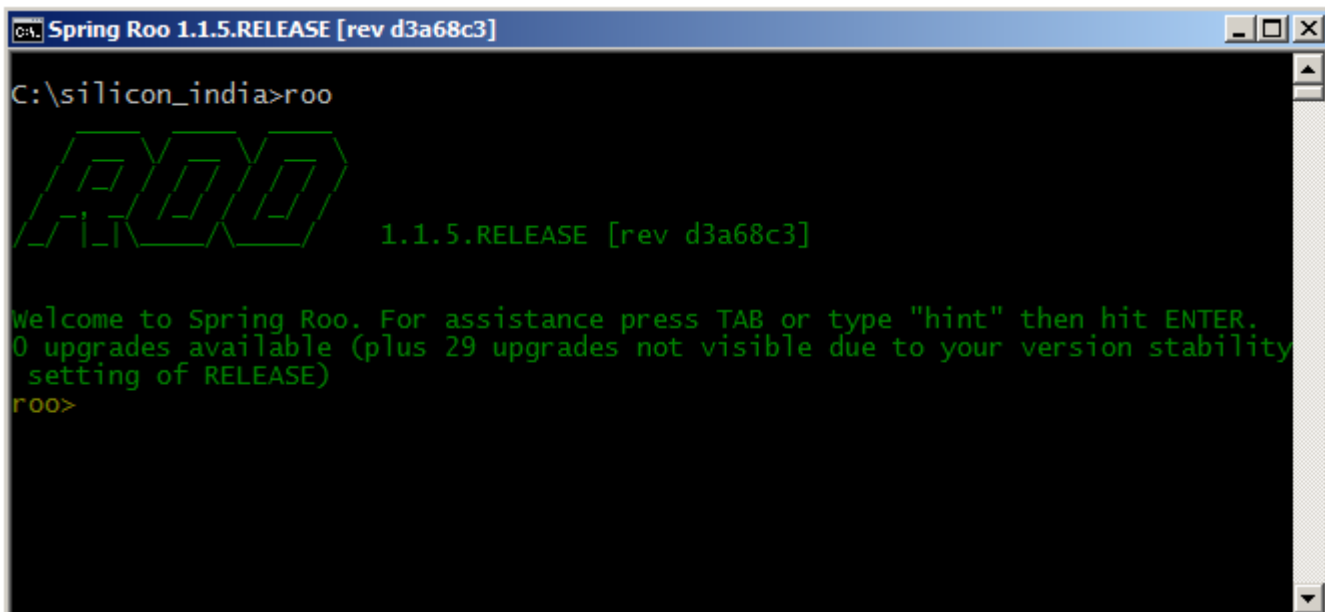
The What and the Why of Spring Roo

- Productivity improvement tool *for developers*
- Code generator (active and passive) for *Spring-based* applications
- Spans *complete lifecycle* of the project
- Applications follow the *best practices* in design
- *Unit and integration tests* can be auto-generated
- Short learning curve – one only needs to know **Spring** and **Maven**.



Spring Roo CLI

- Spring Roo is a *command-line driven* tool
- Spring Roo can be *integrated with IDEs* like STS and Eclipse
- The CLI is *user-friendly*; provides *TAB*-completion features for commands and arguments, and *help* and *hint* commands



```
C:\silicon_india>roo

  AOO  1.1.5.RELEASE [rev d3a68c3]

Welcome to Spring Roo. For assistance press TAB or type "hint" then hit ENTER.
0 upgrades available (plus 29 upgrades not visible due to your version stability
setting of RELEASE)
roo>
```

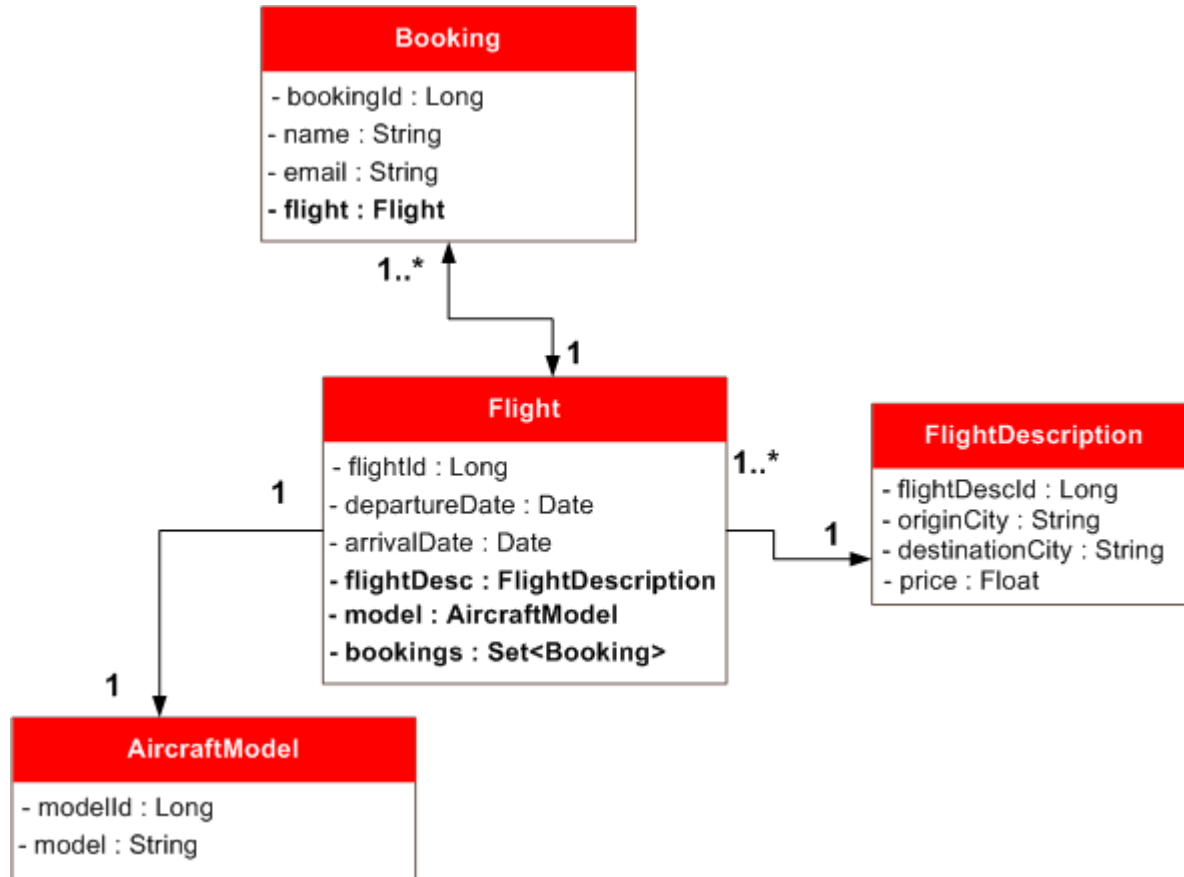
Demo - Flight Booking application

Application Requirements

- Ability to manage **Flight**, **FlightDescription**, **AircraftModel** and **Booking** entities. Ability to search **FlightDescription** instances based on origin and destination fields.
- **Web request security**
 - Web pages for managing **Flight**, **FlightDescription** and **AircraftModel** entities are accessible to users in **ROLE_ADMIN** role
 - Web pages for managing **Booking** entity instances are accessible to users in **ROLE_USER** role
- **Successful creation of a **Booking** entity instance results in *asynchronously* sending a confirmation email to the user**
- **Testing**
 - Integration testing of entities
 - Automated web application testing using Selenium

Demo - Flight Booking application

Domain model



- Many-to-one relationship between **Booking** and **Flight** entities
- Many-to-one relationship between **Flight** and **FlightDescription** entities
- One-to-one relationship between **Flight** and **AircraftModel**

Demo - Flight Booking application

Creating a Roo project

- Roo project is nothing but a Maven project
- Command for creating a Roo project: **project**

```
CA: Spring Roo: com.siliconindia.sample
roo> project --topLevelPackage com.siliconindia.sample --java 6 --projectName flightapp
Created ROOT\pom.xml
Created SRC_MAIN_RESOURCES
Created SRC_MAIN_RESOURCES\log4j.properties
Created SRC_MAIN_RESOURCES\META-INF\spring
Created SRC_MAIN_RESOURCES\META-INF\spring\applicationContext.xml
com.siliconindia.sample roo>
```

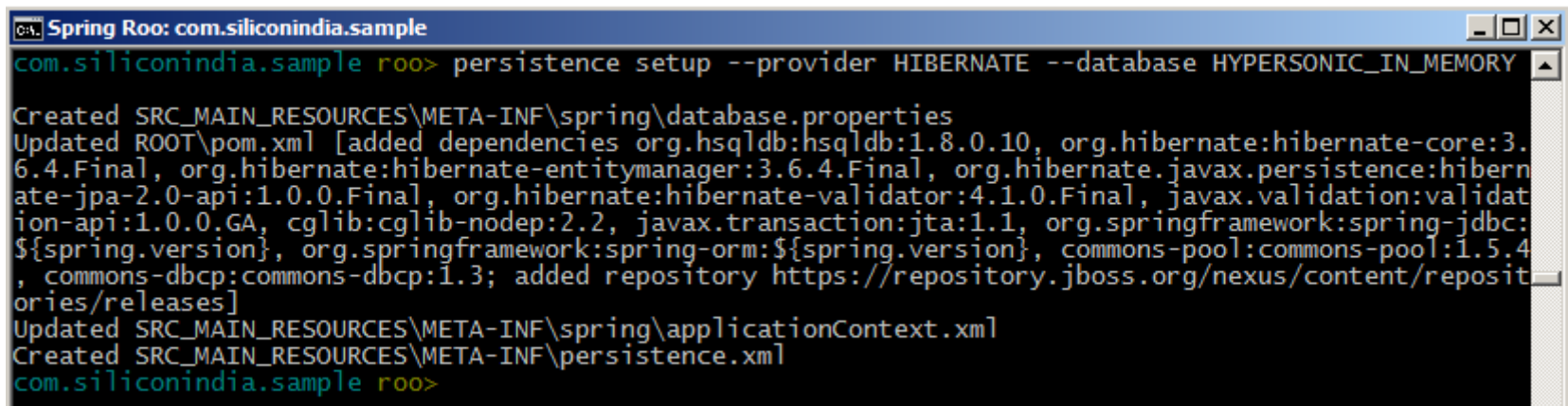
Output from executing **project** command:

- **pom.xml** file : configured with Spring and Spring Roo *annotation* dependencies, Maven Eclipse Plugin, Maven IDEA IDE Plugin, Maven Tomcat Plugin, Maven Jetty Plugin
- **applicationContext.xml** : Spring's application context XML file for the persistence layer.

Demo - Flight Booking application

Setting up a persistence provider

- Roo supports multiple persistence providers (ex. Hibernate, OpenJPA)
- Database information can be configured in a properties file or fetched from JNDI



```
Spring Roo: com.siliconindia.sample
com.siliconindia.sample roo> persistence setup --provider HIBERNATE --database HYPERSONIC_IN_MEMORY

Created SRC_MAIN_RESOURCES\META-INF\spring\database.properties
Updated ROOT\pom.xml [added dependencies org.hsqldb:hsqldb:1.8.0.10, org.hibernate:hibernate-core:3.6.4.Final, org.hibernate:hibernate-entitymanager:3.6.4.Final, org.hibernate.javax.persistence:hibernate-jpa-2.0-api:1.0.0.Final, org.hibernate:hibernate-validator:4.1.0.Final, javax.validation:validation-api:1.0.0.GA, cglib:cglib-nodep:2.2, javax.transaction:jta:1.1, org.springframework:spring-jdbc:${spring.version}, org.springframework:spring-orm:${spring.version}, commons-pool:commons-pool:1.5.4, commons-dbcp:commons-dbcp:1.3; added repository https://repository.jboss.org/nexus/content/repositories/releases]
Updated SRC_MAIN_RESOURCES\META-INF\spring\applicationContext.xml
Created SRC_MAIN_RESOURCES\META-INF\persistence.xml
com.siliconindia.sample roo>
```

Output of executing **persistence setup** command:

- **pom.xml** file is updated to include dependencies corresponding to Hibernate and database driver
- **persistence.xml** and **database.properties** file is created.
- **applicationContext.xml** file is modified to include **DataSource** bean definition.

Demo - Flight Booking application

A few words about the generated Java code

- Roo-generated code is divided into two categories:
 - Code that is managed by Spring Roo – AspectJ ITD files (.aj extension)
 - Code that is managed by developer – Java source files
- At compile-time, code in .aj file is weaved into code contained in Java source file

Demo - Flight Booking application

Creating JPA entities

The **entity** command is used for creating JPA entities:

```
roo> entity --class ~.domain.Flight --identifierColumn FLIGHT_ID --identifierField flightId --identifierType  
java.lang.Long --table FLIGHT_TBL --testAutomatically
```

```
roo> entity --class ~.domain.FlightDescription --identifierColumn FLIGHT_DESC_ID --identifierField flightDescId --  
identifierType java.lang.Long --table FLIGHT_DESC_TBL --testAutomatically
```

Spring Roo supports following features:

- Composite primary key
- Multiple databases
- Mapped superclasses
- Versioning
- Different inheritance strategies (single, joined, table per class)
- Rich entities (CRUD operations are defined in the entity class)

Demo - Flight Booking application

Creating JPA entities (Continued...)

Flight.java

```
@RooJavaBean
```

```
@RooToString
```

```
@RooEntity(identifierField = "flightId", identifierColumn = "FLIGHT_ID", table = "FLIGHT_TBL")
```

```
public class Flight {
```

```
}
```

- **@Roo*** annotations are Roo-specific annotations
- **@Roo*** annotations have Retention policy as **SOURCE**
- **@Roo*** annotations kick-off code generation
- **@RooEntity** informs Roo that Flight is a JPA entity
- **@RooToString** informs Roo to generate toString method for this class
- **@RooJavaBean** informs Roo to generate getters and setters for fields in this class

Demo - Flight Booking application

Creating JPA entities (Continued...)

Flight_Roo_Entity.aj generated corresponding to **@RooEntity** annotation on **Flight.java**

```
privileged aspect Flight_Roo_Entity {  
  
    declare @type: Flight: @Entity;  
  
    declare @type: Flight: @Table(name = "FLIGHT_TBL");  
  
    @PersistenceContext  
    transient EntityManager Flight.entityManager;  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.AUTO)  
    @Column(name = "FLIGHT_ID")  
    private Long Flight.flightId;  
  
    @Version  
    @Column(name = "version")  
    private Integer Flight.version;  
  
    public Long Flight.getFlightId() {  
        return this.flightId;  
    }  
  
    public void Flight.setFlightId(Long id) {  
        this.flightId = id;  
    }  
    ...  
}
```

Demo - Flight Booking application

Adding fields to JPA entities

The **field** commands are used for adding persistent fields to JPA entities:

```
roo> field string --class ~.domain.FlightDescription --fieldName originCity --column FLT_ORIGIN --notNull --sizeMin 3 --sizeMax 20
```

```
roo> field string --class ~.domain.FlightDescription --fieldName destinationCity --column FLT_DESTINATION --notNull --sizeMin 3 --sizeMax 20
```

FlightDescription.java

```
@RooJavaBean
@RooToString
@Entity(identifierField = "flightDescId", identifierColumn = "FLIGHT_DESC_ID", table = "FLIGHT_DESC_TBL")
public class FlightDescription {
```

@NotNull

```
@Column(name = "FLT_ORIGIN")
```

```
@Size(min = 3, max = 20)
```

```
private String originCity;
```

@NotNull

```
@Column(name = "FLT_DESTINATION")
```

```
@Size(min = 3, max = 20)
```

```
private String destinationCity;
```

```
@Column(name = "PRICE")
```

```
private Float price;
```

```
}
```

Demo - Flight Booking application

Adding fields to JPA entities (Continued...)

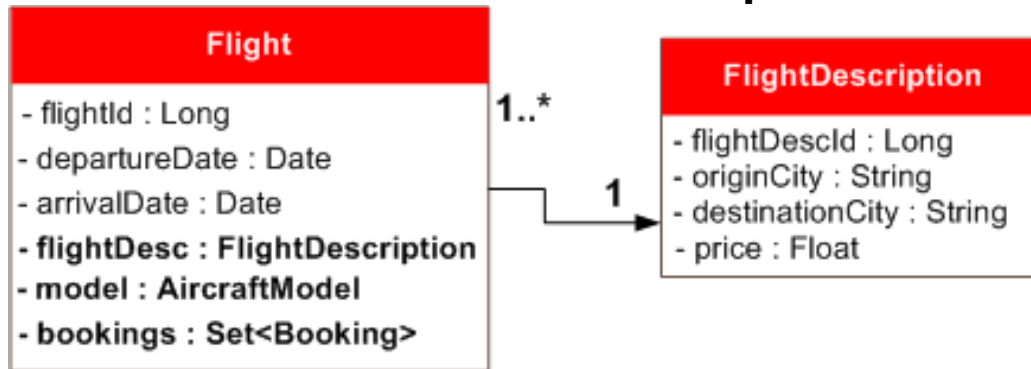
Spring Roo supports following features:

- The field type can be any Java type (**field other** command)
- JSR 303 constraints
- Multiple **field xxx** commands: **field boolean**, **field date**, **field number**, and so on
- **@Embedded** annotated fields (**field embedded** command)
- Relationship fields (**field set** and **field reference** commands)

Demo - Flight Booking application

Creating relationships between entities

The **field reference** command is used for creating the 'many' side of **many-to-one** and 'one' side of **one-to-one** relationship.



```
roo> field reference --class ~.domain.Flight --fieldName flightDescription --type ~.domain.FlightDescription --
joinColumnName FLIGHT_DESC_ID --notNull
```

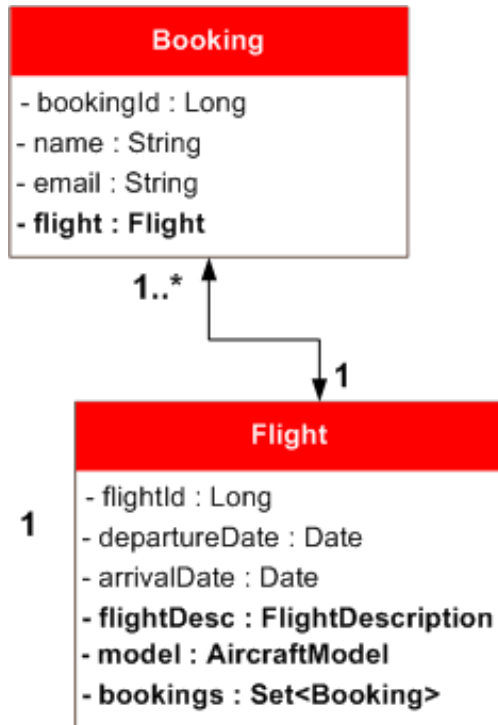
Flight.java

```
@RooJavaBean
@RooToString
@Entity(identifierField = "flightId", identifierColumn = "FLIGHT_ID", table = "FLIGHT_TBL")
public class Flight {
    ...
    @NotNull
    @ManyToOne
    @JoinColumn(name = "FLIGHT_DESC_ID")
    private FlightDescription flightDescription;
    ...
}
```

Demo - Flight Booking application

Creating relationships between entities (Continued...)

The **field set** command is used for creating the 'one' side of **one-to-many** and 'many' side of **many-to-many** relationship.



Flight.java

```
@RooJavaBean
@RooToString
@Entity(identifierField = "flightId", identifierColumn = "FLIGHT_ID", table = "FLIGHT_TBL")
public class Flight {
    ...
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "flight")
    private Set<Booking> booking = new HashSet<Booking>();
    ...
}
```

```
roo> field set --class ~.domain.Flight --fieldName booking --type ~.domain.Booking --cardinality ONE_TO_MANY --mappedBy flight
```


Demo - Flight Booking application

Adding dynamic finder methods

- **finder list** command displays the list of candidate dynamic finder methods
 - The **depth** argument specifies the number of persistent fields to use for listing the candidate dynamic finder methods

```
C:\ Spring Roo: com.siliconindia.sample
~.domain.Flight roo> finder list --class com.siliconindia.sample.domain.FlightDescription
findFlightDescriptionsByDestinationCityEquals(String destinationCity)
findFlightDescriptionsByDestinationCityIsNotNull()
findFlightDescriptionsByDestinationCityIsNull()
findFlightDescriptionsByDestinationCityLike(String destinationCity)
findFlightDescriptionsByDestinationCityNotEquals(String destinationCity)
findFlightDescriptionsByOriginCityEquals(String originCity)
findFlightDescriptionsByOriginCityIsNotNull()
findFlightDescriptionsByOriginCityIsNull()
```

- **finder add** command is for adding a dynamic finder method to an entity

```
C:\ Spring Roo: com.siliconindia.sample
~.domain.FlightDescription roo> finder add --class ~.domain.FlightDescription findFlightDescriptions
ByDestinationCityLikeAndOriginCityLike
Updated SRC_MAIN_JAVA\com\siliconindia\sample\domain\FlightDescription.java
Created SRC_MAIN_JAVA\com\siliconindia\sample\domain\FlightDescription_Roo_Finder.aj
~.domain.FlightDescription roo>
```

Demo - Flight Booking application

Scaffolding Spring Web MVC application

- **controller all** command scaffolds JSPX views and Spring Web MVC controllers corresponding to JPA entities

```
roo> controller all --package ~.web
```


Spring Roo support following features:

- Scaffold Spring Web MVC controller and JSPX views corresponding to a specific JPA entities (**controller scaffold** command)
- Manually create a Spring Web MVC controller (**controller class** command)
- RESTful web controllers (based on HTTP **GET**, **POST**, **PUT** and **DELETE**)
- Internationalization of messages and labels
- Themes
- Custom tag library (**tagx** files)
- Apache Tiles 2 framework

Demo - Flight Booking application

Scaffolding Spring Web MVC application (Continued...)

ROO



FLIGHT DESCRIPTION

- Create new Flight Description
- List all Flight Descriptions
- Find by Destination City Like And Origin City Like

AIRCRAFT MODEL

- Create new Aircraft Model
- List all Aircraft Models

FLIGHT

- Create new Flight
- List all Flights


BOOKING


- Create new Booking
- List all Bookings

▼ Welcome to Flightapp

Welcome to Flightapp

Spring Roo provides interactive, lightweight and user customizable tooling that enables rapid delivery of high performance enterprise Java applications.

[Home](#) | Language:  | Theme: [standard](#) | [alt](#)

Sponsored by SpringSource 

Demo - Flight Booking application

Adding security to your application

- **security setup** command adds **Spring Security** to your application

```
roo> security setup
```

Result of executing **security setup** command:

- An **applicationContext-security.xml** file is created which configures Spring Security
- A **login.jspx** view is created which renders the login page of the application
- The **web.xml** file is updated to use Spring Security filter for intercepting access to secured resources
- Dependencies related to Spring Security are added to **pom.xml** file

Customizing Roo-generated configuration

Spring Security configuration

- **Change web request security configuration:**

```
<intercept-url pattern="/flightdescriptions/**" access="hasRole('ROLE_ADMIN')"/>
<intercept-url pattern="/aircraftmodels/**" access="hasRole('ROLE_ADMIN')"/>
<intercept-url pattern="/flights/**" access="hasRole('ROLE_ADMIN')"/>
<intercept-url pattern="/bookings/**" access="hasRole('ROLE_USER')"/>
```

- **Change authentication provider:**

```
<authentication-manager>
  <authentication-provider>
    <ldap-user-service group-search-filter="..." group-search-base="..." user-search-base="..."
      user-search-filter="..." group-role-attribute="..." />
  </authentication-provider>
</authentication-manager>
```

- **Add method-level security:**

```
<global-method-security mode="aspectj" secured-annotations="enabled"/>
```

Sending emails

- **email sender, email template setup and field email template commands are used to add JavaMail support to your application**

```
roo> email sender setup --hostserver xyz
```

```
roo> email template setup --from someone@siliconindia.com
```

```
roo> field email template --class ~.web.BookingController --async
```

Result of executing above commands:

- A **sendMessage** method is added to **BookingController** class which uses Spring's **MailSender** to send emails
- Email template (from, to) and server (host, username, password) properties are configured in an **email.properties** file
- Spring's **MailSender** is configured in **applicationContext.xml**

Customizing Roo-generated code

Sending emails

Problem: Modifying **create** method defined in AspectJ ITD file

```
privileged aspect BookingController_Roo_Controller {
  @RequestMapping(method = RequestMethod.POST)
  public String BookingController.create(@Valid Booking booking, ...) {
    if (bindingResult.hasErrors()) {
      uiModel.addAttribute("booking", booking);
      return "bookings/create";
    }
    uiModel.asMap().clear();
    booking.persist();
    //-- email sending code should come here
    return "redirect:/bookings/" + encodeUrlPathSegment(booking.getBookingId().toString(), httpServletRequest);
  }
  ...
}
```

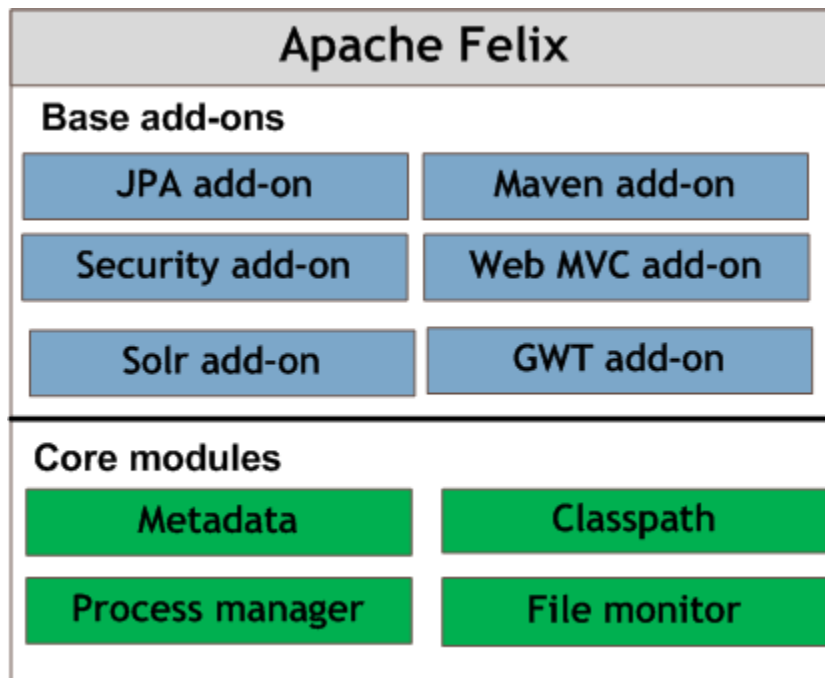
Solution: Define the **create method in **BookController.java** or perform **Push-in refactoring****

What more you can do with Spring Roo

- **Create JPA layer from an existing database using Database Reverse Engineering commands**
- **Create Flex, Spring Web MVC, GWT and Spring Web Flow applications**
- **Create applications Spring Web MVC and GWT applications for GAE**
- **Create applications for Cloud Foundry**
- **Add JMS support to your applications**
- **Create Selenium tests**
- **Add Solr support to your applications**

Spring Roo architecture

- Built around **Apache Felix OSGi container**
- Spring Roo consists of **base add-ons** and **core modules**



Extending Roo with custom add-ons

- Create custom add-ons with **Add-on Creator** add-on
- Option to build **simple** and **advanced** add-ons
 - **Simple add-on** is for creating configuration or modifying files and for copying artifacts
 - **Advanced add-on** is when you want to generate Java code and AspectJ ITDs
- Add-on developer works with the utility classes and services provided by Spring Roo
- Developers can also install available **installable add-ons** from RooBot or any other repository

The road ahead

- The latest production release is 1.1.5
- The latest milestone release is 1.2.0.M1
- Integration with **MongoDB** available in 1.2.0.M1
- A **service** command is available in 1.2.0.M1 to create application's service layer
- Roo shell is 10 times faster in 1.2.0.M1
- **JSF/PrimeFaces** add-on is planned
- **Multi-module Maven** support is planned

Shameless promotion



Thank you.