**RADSpringBootGen**

Spring Boot Application Code Generation Framework

Technical Documentation

Draft

By Howard Hyde

Last updated 2018.08.26

*Write to* [*hhyde007@icloud.com*](mailto:hhyde007@icloud.com) *with your questions, comments and suggestions on how to improve this document and the framework itself.*

Table of Contents

[Part One: Introduction and Tutorial 4](#_Toc523057747)

[Introducing RADSpringBootGen 4](#_Toc523057748)

[Sample Application: “FBOAce” 5](#_Toc523057749)

[The UI and Input 7](#_Toc523057750)

[Example Model Entity: Flight 8](#_Toc523057751)

[The Output 10](#_Toc523057752)

[How did it do that, or…Where did that come from? 13](#_Toc523057753)

[Sample App: One More Thing Before You… 13](#_Toc523057754)

[Try it Yourself 16](#_Toc523057755)

[The RADSpringBootGen project imported into your SpringSTS workspace. 16](#_Toc523057756)

[A database instance accessible from your running Spring Boot apps 16](#_Toc523057757)

[A Database Driver File 16](#_Toc523057758)

[An **application.properties** file with the correct configuration entries for the database access. 16](#_Toc523057759)

[Let’s run it! 17](#_Toc523057760)

[Now Build YOUR Application 18](#_Toc523057761)

[Part Two: Customizing the Template 21](#_Toc523057762)

[“I like it, BUT…” 21](#_Toc523057763)

[The Service Interface Template 21](#_Toc523057764)

[Service.java.template.txt 22](#_Toc523057765)

[What you are seeing 23](#_Toc523057766)

[Nested/Multivalued tokens: putEntity() 27](#_Toc523057767)

[deleteEntity() 30](#_Toc523057768)

[Template Comments 30](#_Toc523057769)

[Delete Child Instances 30](#_Toc523057770)

[Token, token, Who’s got the Token? 32](#_Toc523057771)

[Part Three: Enhancing RADSpringBootGen’s Functionality 38](#_Toc523057772)

[Internal Technical Documentation 38](#_Toc523057773)

[Launch! 40](#_Toc523057774)

[[TO BE CONTINUED…] 40](#_Toc523057775)

# Part One: Introduction and Tutorial

## Introducing RADSpringBootGen

**RADSpringBootGen** (hereinafter referred to as “The Framework”) is a Java Spring Boot-based tool for generating first-cut classes and interfaces for Spring Boot applications, having first in mind (but not limited to) interactive OLTP relational database applications using a Thymeleaf HTML template-based web user interface .

* YOU WRITE: Database table and view definitions (SQL DDL), JPA/Hibernate-annotated model entity classes, repository interfaces.
* RADSpringBootGen GENERATES:
  + Service interfaces
  + Service implementation classes
  + Web controller classes
  + HTML UI (Thymeleaf template) data maintenance, editing and navigation forms.

…so that you don’t have to write these manually.

*NOTE: A certain basic familiarity with Java, Spring Boot and relational databases is assumed. This is NOT a tutorial on Spring Boot itself or the SpringSTS framework, based on Eclipse. If you need background information on those, see the documentation at https://****spring.io****/projects/spring-boot and/or https://****spring.io****/guides/gs/spring-boot. Our favorite tutorial on Spring Boot is the YouTube series by* ***JavaBrains****, starting with https://www.youtube.com/watch?v=msXL2oDexqw.*

*For information on using the* ***Thymeleaf*** *HTML template web page framework with Spring, see https://****www.thymeleaf.org****/doc/tutorials/2.1/thymeleafspring.html.*

*For information about the* ***Bootstrap*** *stylesheet framework, see https://getbootstrap.com.and https://****getbootstrap.com****/docs/4.0/getting-started/introduction.  
For more information about using Bootstrap and Thymeleaf with Spring Boot in database applications, see Marcio Marinho’s tutorial “Building a CRUD Web Application with Spring Boot” at https://www.youtube.com/watch?v=TcP5kFPq354*

*Likewise, installation guidance and introductory tutorials on relational databases are beyond the scope of this document. Consult your technology-specific documentation, training materials and online resources.*

Like Spring Boot itself, RADSpringBootGen is *opinionated*, in that it assumes certain conventions and defaults are to be followed, rather than necessarily providing for every imaginable (or unimaginable) variation.

A web database application generated by RADSpringBootGen follows the architecture and conventions enumerated below. Items marked with an asterisk\* are the components that RADSpringBootGen generates automatically by reading the lower-level objects.

* A database schema of tables that follows relational normalization design conventions. **Oracle** is the only currently supported database. Knowledgeable persons wishing to work with other databases are encouraged to learn the internals and contribute to this project.
* Data model/entity bean classes annotated according to JPA/Hibernate conventions to represent the database tables, columns, keys and relationships in the application.
* Repository interfaces that extends org.springframework.data.repository.CrudRepository or \*.JpaRepository corresponding to each of the annotated model entities.
* \*Service Java interfaces
* \*ServiceImpl Java classes, annotated as @Service, that implement business logic and processing between the (web) controllers and the repositories.
* \*Web controller Java classes, annotated as @Controller, which implement the logic of the servlet, serving incoming HttpRequests and making calls to the Service layer.
* \*Thymeleaf template HTML files for web user interface data maintenance forms. Please note the common static HTML file ***list.html*** that is shared among all modules of the generated application as the generic entry point, listing all of the records in the table (or whatever subset the Repository provides).

## Sample Application: “FBOAce”

Throughout the exposition, you can learn a lot by exploring the sample database application “***FBOAce***”, a stripped-down make-believe Flight-Aircraft-Pilot scheduling system (“FBO” stands for “Fixed Base Operator,” a business that provides aircraft rental, fuel, maintenance and pilot training at an airport). Sample application source code files are in the directory com\radinfodesign\RADSpringBootGen\fboace\**model** off of the RADSpringBootGen project root (separate from src, target etc.). The SQL script to create and populate the sample database application tables and other objects is under the project root at demo-db-schema/**export-fboace03a.sql** . Sample application model entity classes (which get compiled and deployed to the target directory along with the rest of the RADSpringBootGen application) are stored under RADSpringBootGen project root in src\main\java\com\radinfodesign\RADSpringBootGen\fboace\**model**, about which more anon[[1]](#footnote-1).

Here is the data model for FBOAce (version 3), which informs the narrative about any and all entities and relationships:

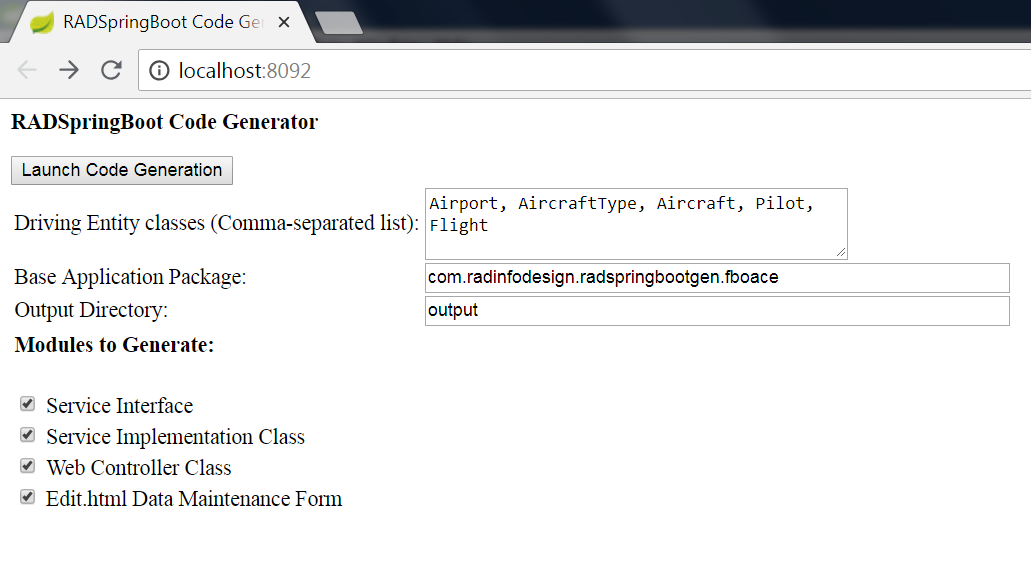
A close up of text on a white background

Description generated with high confidence

## The UI and Input

*[NOTE: Professional driver on closed course. Do not attempt until we have explained prerequisites, setup and launch of the framework just a little later.]*

When you fire up RADSpringBootGen from within SpringSTS on your local machine and browse it at the indicated (customizable) port, you see something like this:



The values in the text boxes (Airport, AircraftType etc., com.radinfodesign etc., output) and the state of the checkboxes, are *defaults*, hard-wired on the assumption that the demonstration application and its annotated model entities and repositories are present and that you want to generate the corresponding service interfaces, service implementation classes, web controller classes and/or HTML data maintenance forms. When you customize the framework for your own use, you will be free to override these values here or even change the hard-wired default values to suit your requirements.

Again, when we list Airport, AircraftType, Aircraft, Pilot and/or Flight as entities for which we wish to generate service interfaces, service implementation classes, web controller classes and/or HTML data maintenance forms, implicitly it means that JPA/Hibernate-annotated model entity classes of those names are defined in .java files in the /model directory under the base application package location. Any misspelling or missing .java files could lead to runtime errors.

### Example Model Entity: Flight

**Flight.java** is a good example of a rich data model entity, with simple attributes and multiple references to ‘parent’ entities (Aircraft and Airport, the latter in two distinct roles; one as Departure, the second as Destination). It also keeps a collection of instances of a child entity, FlightCrewMember, which itself is essentially a cross-reference between one instance of a Flight entity and one instance of an instance of Pilot (Many-to-Many, a.k.a. M:N relationship; a giving instance of Flight may be associated with one or more instances of Pilot, and a Pilot may be associated with one or more instances of Flight). Explore the sample application \fboace\model directory for other examples.

The listing also illustrates most of the imports that are required to support the JPA annotations which make the correspondence between Java elements and database objects and components work.

*(Getter and Setter methods excluded)*

/\*

\* Sample application model entity class for RADSpringBootGen

\* Copyright(c) 2018 by RADical Information Design Corporation

\* Flight

\*/

package com.radinfodesign.RADSpringBootGen.fboace.model;

import java.io.Serializable;

import java.util.Collection;

import java.time.LocalDateTime;

import javax.persistence.Basic;

import javax.persistence.CascadeType;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.Id;

import javax.persistence.JoinColumn;

import javax.persistence.ManyToOne;

import javax.persistence.NamedQueries;

import javax.persistence.NamedQuery;

import javax.persistence.OneToMany;

import javax.persistence.SequenceGenerator;

import javax.persistence.Table;

import javax.persistence.Temporal;

import javax.persistence.TemporalType;

import javax.xml.bind.annotation.XmlRootElement;

import javax.xml.bind.annotation.XmlTransient;

import com.radinfodesign.RADSpringBootGen.model.Label;

/\*\*

\*

\* @author Tarzan

\*/

@Entity

@Table(name = "FLIGHT")

@XmlRootElement

public class Flight implements Serializable {

private static final long serialVersionUID = 1L;

// @Max(value=?) @Min(value=?)//if you know range of your decimal fields consider using these annotations to enforce field validation

@Id

@Basic(optional = false)

@Column(name = "FLIGHT\_ID")

@GeneratedValue(generator="InvSeq")

@SequenceGenerator(name="InvSeq", sequenceName="FLIGHT\_PK\_SEQ")

private Integer flightId;

@Basic(optional = false)

@Column(name = "SHORT\_NAME")

private String shortName;

@Column(name = "LONG\_NAME")

private String longName;

@Column(name = "DEPARTURE\_DATE\_TIME")

private LocalDateTime departureDateTime;

@Column(name = "ARRIVAL\_DATE\_TIME")

private LocalDateTime arrivalDateTime;

@Column(name = "NOTES")

private String notes;

@JoinColumn(name = "AIRCRAFT\_ID", referencedColumnName = "AIRCRAFT\_ID")

@ManyToOne(optional = false)

@Label (name="Aircraft")

private Aircraft aircraftId;

@JoinColumn(name = "AIRPORT\_ID\_DEPARTURE", referencedColumnName = "AIRPORT\_ID")

@ManyToOne

@Label (name="Departure Airport")

private Airport airportIdDeparture;

@JoinColumn(name = "AIRPORT\_ID\_DESTINATION", referencedColumnName = "AIRPORT\_ID")

@ManyToOne

@Label (name="Destination Airport")

private Airport airportIdDestination;

@OneToMany(cascade = CascadeType.PERSIST, mappedBy = "flight")

private Collection<FlightCrewMember> flightCrewMemberCollection;

public Flight() {

}

public Flight(Integer flightId) {

this.flightId = flightId;

}

public Flight(Integer flightId, String shortName) {

this.flightId = flightId;

this.shortName = shortName;

}

@Override

public int hashCode() {

int hash = 0;

hash += (flightId != null ? flightId.hashCode() : 0);

return hash;

}

@Override

public boolean equals(Object object) {

// TODO: Warning - this method won't work in the case the id fields are not set

if (!(object instanceof Flight)) {

return false;

}

Flight other = (Flight) object;

if ((this.flightId == null && other.flightId != null) || (this.flightId != null && !this.flightId.equals(other.flightId))) {

return false;

}

return true;

}

@Override

public String toString() {

return this.shortName;

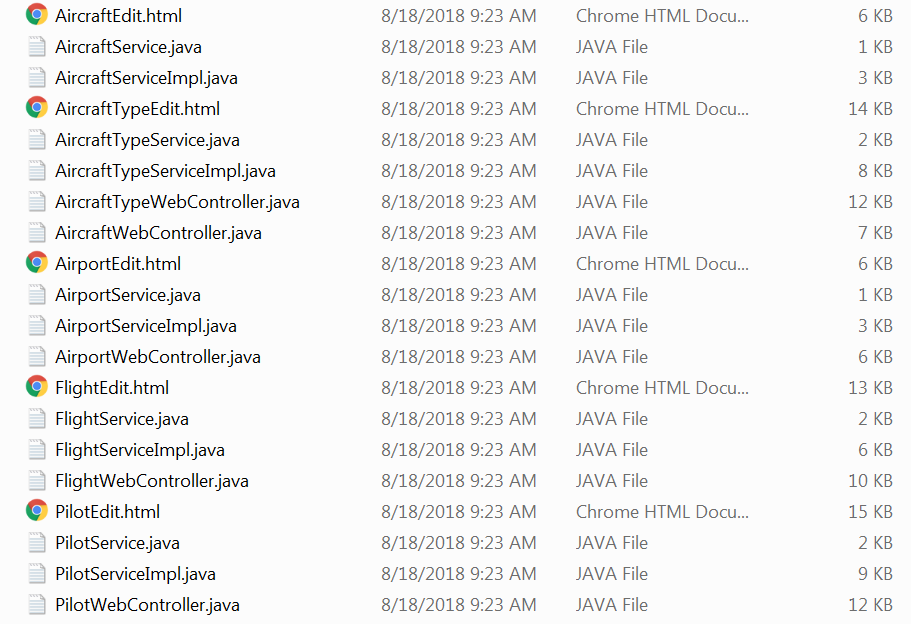
}

}

## The Output

Back to the code generator web page: When you click on the **[Launch Code Generation]** button… nothing happens (except you might see a change of URL in the address bar).

Nothing *appears* to happen, but if all artifacts are in place and the stars align, when you check the output directory indicated, you will see something like the following:



All of the specified files were generated for you and saved to the output directory.

Let’s look at a few of the *simplest* generated files to get an idea of what’s going on here. Consider the following Service interfaces (*copyright notice and other comments removed, whitespace compressed*):

**AirportService.java**

package com.radinfodesign.RADSpringBootGen.fboace.service;

import java.util.List;

import com.radinfodesign.RADSpringBootGen.fboace.model.Airport;

public interface AirportService {

List<Airport> getAll();

Airport getEntity (Integer airportId);

Airport putEntity

( Integer airportId

, String shortName

, String iataCode

, String description

, String portType

) throws Exception;

int deleteEntity(Integer airportId);

}

**AircraftService.java**

package com.radinfodesign.RADSpringBootGen.fboace.service;

import java.util.List;

import com.radinfodesign.RADSpringBootGen.fboace.model.Aircraft;

public interface AircraftService {

List<Aircraft> getAll();

Aircraft getEntity (Integer aircraftId);

Aircraft putEntity

( Integer aircraftId

, String shortName

, String longName

, String description

, String notes

, Integer aircraftTypeId

) throws Exception;

int deleteEntity(Integer aircraftId);

}

**FlightService.java**

package com.radinfodesign.RADSpringBootGen.fboace.service;

import java.util.List;

import com.radinfodesign.RADSpringBootGen.fboace.model.Flight;

public interface FlightService {

List<Flight> getAll();

Flight getEntity (Integer flightId);

Flight putEntity

( Integer flightId

, String shortName

, String longName

, String departureDateTime

, String arrivalDateTime

, String notes

, Integer aircraftId

, Integer airportIdDeparture

, Integer airportIdDestination

, String[] flightCrewMemberNotess

, Integer[] flightCrewMemberPilotIds

) throws Exception;

int deleteEntity(Integer flightId);

int deleteFlightCrewMember

( Integer flightCrewMemberFlightId

, Integer flightCrewMemberPilotId

);

List<Flight> getQualifiedFlightsByPilotId (Integer pilotId);

}

These three Service interfaces represent three related entities (four more exist in the sample application: AircraftType, Pilot, PilotCertification and FlightCrewMember). Each of these Service interfaces allow the caller (which in our architecture is the web controller class) to retrieve a list collection of instances of the class; retrieve an individual instance based on a primary key value; to post back the attribute values of a new or existing instance, i.e. to initiate a SQL INSERT or UPDATE on the back end (the Service *Implementation* *class* marshals the instance from the parameters and delegates the database work to the JPA/Hibernate-annotated Repository); and to delete an individual instance based on a primary key value.

The example FlightService also includes a method for deleting individual child records: records in tables which reference the present one (Flight) via foreign keys, in this case FlightCrewMember. It also exposes a semi-custom method called getQualifiedFlightsByPilotId().

The point of showing you these Service interfaces is that

1. They are all different, and
2. They are all the same.

They are all different because they contain elements that are specific to a particular data model entity class and no other, and they are all the same because they all follow common *patterns* of structure and presentation. And because they all follow common patterns, we can abstract the pattern into a *template* from which to automatically generate the individual source code files and their specific content. Compare each of the service interfaces with the model entity class that it is based upon to see the correlation.

## How did it do that, or…Where did that come from?

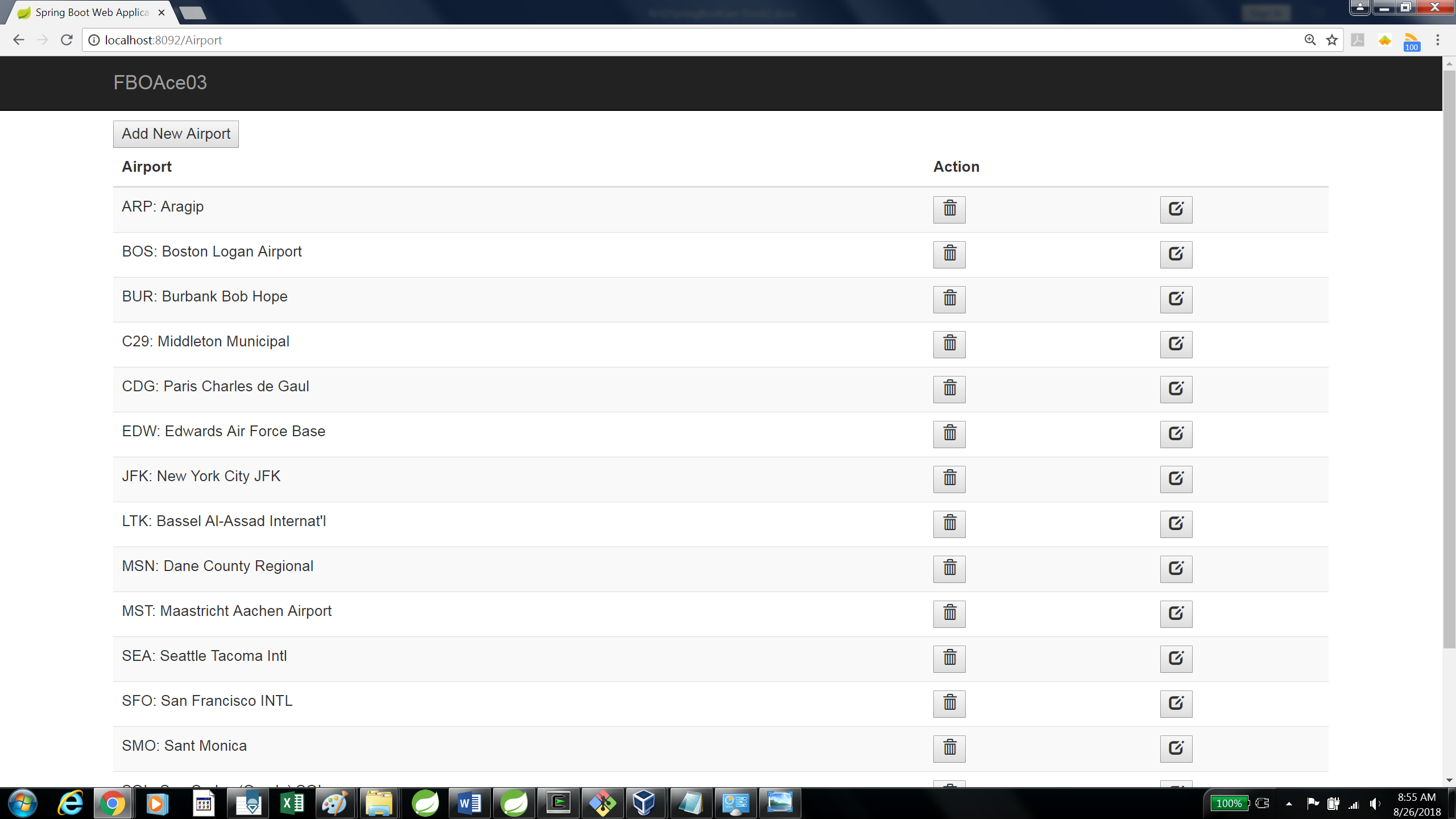
As of the current version (writing August 2018), RADSpringBootGen reads the previously-written JPA/Hibernate-annotated source code files for the model entities and corresponding metadata from the actual database tables, loads the model entity classes into memory for reading via reflection at runtime, parses a template file, and with all of that information is able to automatically generate the code for the above Service interfaces, or any of a million others that follow the same conventions.

As noted above, the framework can generate Service interfaces, Service Implementation classes, Web Controller classes and Thymeleaf HTML data maintenance forms – or theoretically *anything* that may be derived or inferred by the database table metadata and/or annotated model entity class definitions. We start with the Service interface because it is the simplest of the lot.

## Sample App: One More Thing Before You…

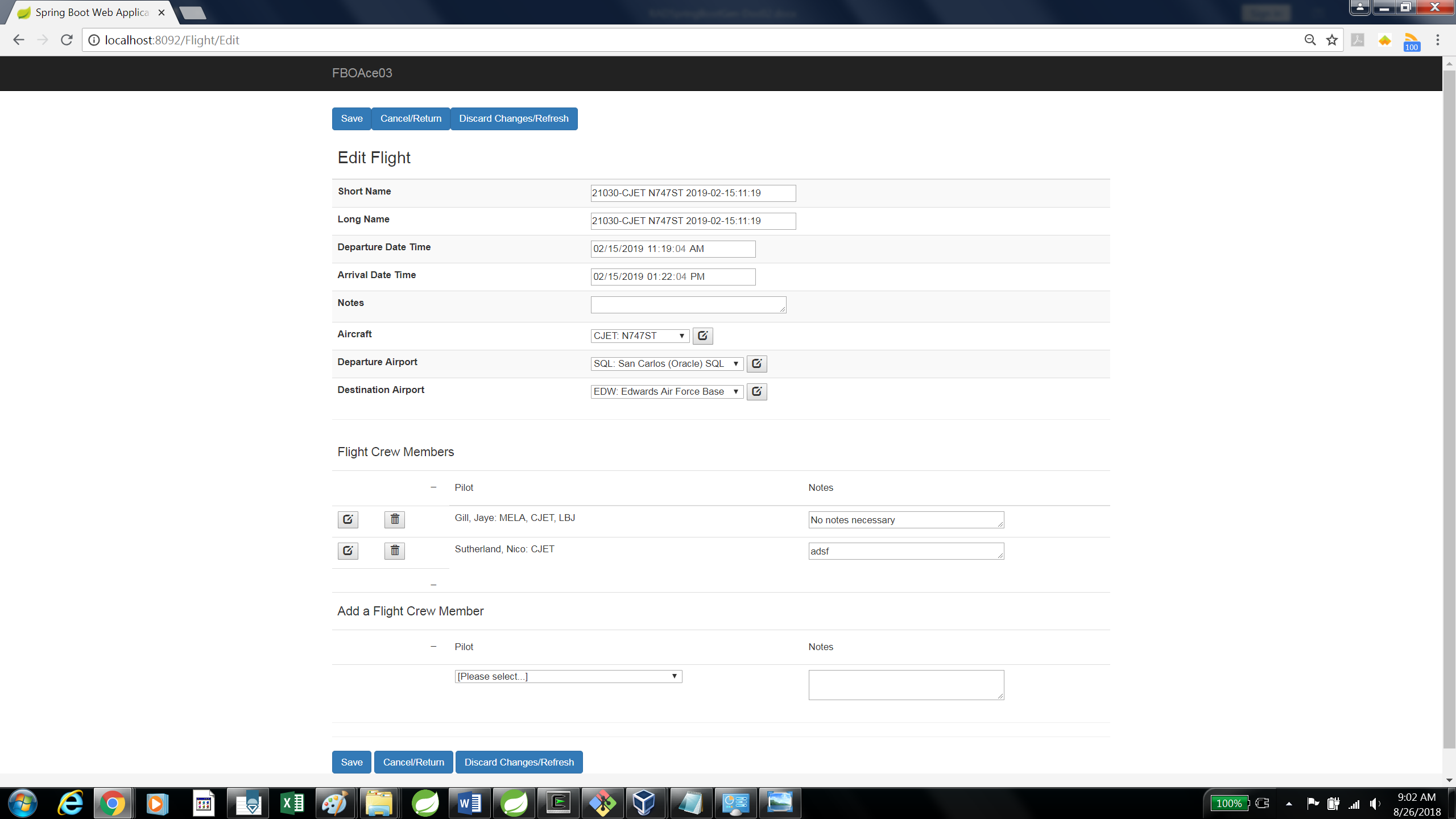
We showed above how you can browse the running RADSpringBootGen app at http://localhost:8092 and see the default list of entities for which you may generate code. But the framework holds one more secret. Try appending one of those entity names in the list to the URL – any one of them, as in:

* http://localhost:8092/**Airport**
* http://localhost:8092/**AircraftType**
* http://localhost:8092/**Aircraft**
* http://localhost:8092/**Pilot**  
  or
* http://localhost:8092/**Flight**



Hello, what’s this? **The demo database application is built into the code generation framework**. From any one of the “List” forms that are presented by the above URLs, you may select on instance of that entity to drill into.

For example, in one version of the demo app, if you enter the URL http://localhost:8092/Flight and select the Flight instance with the name/identifier “21030-CJET N747ST 2019-02-15:11:19”, you get this ***Edit*** form:



*[By the way, you may send your flaming comments on our graphical UI design to the (non-existent) Complaint Department. Visual art is not our forte here. The framework provides the data and functionality infrastructure over which your artistic talent with HTML, CSS, JavaScript derivatives etc. may thrive. As we will see later, you will have the ability to customize the templates, which will give you the opportunity to incorporate visual design elements as you require.]*

The app has enough features that we could spend the next 10 pages or more detailing them, but these are best discovered through exploration. Let’s just summarize by saying that, depending upon each entity’s (table’s) relationships to other entities, the user may:

* *Add* (SQL INSERT) new instances of the main (or “driving”) entity.
* *Edit* (SQL UPDATE) attributes/fields of the instance of the driving entity, including choosing validated references to instances of other/parent entities via drop-down list boxes.
* *Edit* attributes of instances of other entities which are children or multi-valued detail records of the driving entity instance.
* *Add* (SQL INSERT) new instances of child entities.
* *Navigate* to forms where a referenced parent, child or “third” entity instance is the driving entity instance, for example from the edit form for Flight “21030-CJET N747ST 2019-02-15:11:19” to the form for Airport “SQL: San Carlos (Oracle)”; or to the edit form for Pilot “Nico Sutherland”, who in this example holds a Pilot Certification in “CJET” or “Commercial Jet Airplane”.

Please spend the next hour exploring what you can do with the sample application, understanding that the framework can generate the same functionality for YOUR database application, and then…

## Try it Yourself

With that understanding then, give the code generator a try yourself, first using the default demo database app. Here are the **prerequisites**:

### The RADSpringBootGen project imported into your SpringSTS workspace.

Downloaded the complete RADSpringBootGen project from <https://github.com/hhyde007/RADSpringBootGen>.

Consult the Spring Boot and SpringSTS documentation for importing and running projects.

### A database instance accessible from your running Spring Boot apps

Log into to your (Oracle) database as a user with DBA privileges and run the script demo-db-schema/**export-fboace03a.sql** to create the database user/schema and create and sample-populate all of the requisite objects – tables, views, triggers etc.

### A Database Driver File

For example **ojdbc7-12.1.0.1.jar** or equivalent, downloaded from https://www.oracle.com/technetwork/database/application-development/jdbc/overview/quickstart-4308895.html. and copied to the src\main\resources directory of your RADSpringBootGen project, or better yet to your common repository directory (independent of specific SpringSTS/Eclipse projects), for example C:\Users\Tarzan\.m2\repository\com\oracle\ojdbc7\12.1.0.1.

### An **application.properties** file with the correct configuration entries for the database access.

The sample one has the following entries, which you may need to tweak (customize to your environment):

spring.datasource.url=jdbc:oracle:thin:@192.168.0.12:1539/PDBRad78

spring.datasource.username=hhyde\_fboace03\_oltp\_tab

spring.datasource.password=fboace3

spring.datasource.driver-class-name=oracle.jdbc.OracleDriver

spring.datasource.tomcat.max-active=10

spring.jpa.database-platform=org.hibernate.dialect.Oracle12cDialect

### Let’s run it!

With the prerequisites met, you should be able to fire up RADSpringBootGen, connect to the database automatically and be off and running.

Open your SpringSTS where you have imported the RADSpringBootGen project, find **RADSpringBootGenApplication.java** under src\main\java\com\radinfodesign\RADSpringBootGen, right-click it and choose Run as…/Spring Boot Application. The app should boot up (put intended) in a few seconds and give you a message in the Console window similar to the following:

2019-07-08 14:21:55.523 INFO 4988 --- [ main] s.b.c.e.t.TomcatEmbeddedServletContainer : **Tomcat started on port(s): 8092** (http)

2019-07-08 14:21:55.526 INFO 4988 --- [ main] c.r.r.RADSpringBootGenApplication : Started RADSpringBootGenApplication in 3.72 seconds (JVM running for 4.393)

What could possibly go wrong?

The most common source of problems booting up RADSpringBootGen is database connectivity. Make sure your JDBC driver is installed correctly and the JDBC connect string and login credentials in the application.properties file are all perfect.

At this point, you’ll want to circle back to “[The UI and Input](#_The_UI_and)” above and confirm that you can reproduce what we demonstrated to you in the last several pages. With RADSpringBootGen running, open your browser and enter the URL in the address bar, for example http://localhost:8092. The web form we showed you above should appear. Make sure that your /output directory is empty; delete or move any files there to a /bak sub-directory.

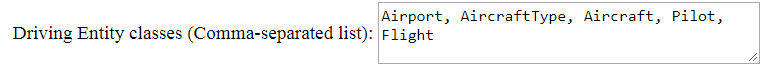
Try generating code for the default list of entities, and for all modules, according to the defaults. Verify that the intended files were generated and saved to the /output directory. Delete or make backup copies of those files. Try again with a subset of the entities or just one, and a subset of the module check boxes checked, or just one, or in any combination you choose. Your results should be consistent with your intentions.

Congratulations, you are now a Master RADSpringBootGen Operator.

## Now Build YOUR Application

Now that you have a feel for how the framework works, here are the steps to generate the code for your database application. We assume that the same prerequisites as above are satisfied.

* **Create a database schema**/user account in your (Oracle) database accessible from your app. Again, for an example of this and the next step, see the SQL script file demo-db-schema/**export-fboace03a.sql**.
* **Create the objects**: Tables and (optional but recommended) views which represent them
  + It is a good idea to pre-populate reference/Type tables via scripted SQL DML (INSERT) statements.
  + *Note that for part of the code generation process, RADSpringBootGen will read metadata about these tables directly by logging into the database account and querying the USER\_TABLES and USER\_TAB\_COLUMNS (and possibly other) metadata views.*
* Edit the application.properties file database configuration entries as necessary.
* Clone the database application packages: Create a custom Java package (and corresponding directory structure) under com.radinfodesign.RADSpringBootGen, which is peer to the demo app package fboace (which may be deleted if you never intend to use it again).  
  These classes will be compiled and read via reflection during the code generation process.
  + REQUIRED: Under your custom package, create the sub-packages .model and .dao.
  + OPTIONAL: Under your custom package, create the sub-packages .service and .controller.
  + *NOTE: In a future version of RADSpringBootGen, the following steps may also be automated:*
  + Under .model, create or generate the JPA and/or Hibernate-annotated Java classes that represent your database tables to the app. For examples of these, see the .java files under com\radinfodesign\RADSpringBootGen\fboace\model.  
    *(Most Java IDEs have a utility to generate these classes by reading metadata from a database. We have had the best results so far with the one provided by NetBeans. Some manual touch-up will likely be required. Again, see the examples.)*
  + Under .dao, create the Repository interfaces necessary for your Service layer to access and persist the model entities. See com\radinfodesign\RADSpringBootGen\fboace\dao for examples.
* Clone the same database application package/directory structure and .java files (only the /model final package files, *not necessary /dao, /service* or */controller*), copying to a structure off the root of the project. You will notice that there is a sample-application /fboace directory under com\radinfodesign\RADSpringBootGen off of the root of the project directory in addition to the one within the project under src\main\java\com\radinfodesign\RADSpringBootGen.  
  The .java files here get parsed as text files during code generation, independently of their counterparts within the project being compiled, loaded and read via reflection. Note that the .java files under /src/main/com etc. are NOT copied to the /target directories when the project is built – only their corresponding .class files are – and are therefore not available to the application as source code text files at runtime.
* OPTIONAL: Edit the HTML file src\main\resources\templates\app\**index.html** to change the hard-coded default values presented to the user of the RADSpringBootGen web app (delete the FBOAce entity class names and replace with your own).  
  For example, this form element:



…is the result of this hard-coded HTML:

<tr>

<td>

<label for=*"entityList"*>

Driving Entity classes (Comma-separated list):

</label>

</td>

<td>

<textarea rows=*"3"* id=*"entityList"* name=*"entityList"* cols=*"40"*>

Airport, AircraftType, Aircraft, Pilot, Flight

</textarea>

</td>

</tr>

* Stop, rebuild/recompile and run the RADSpringBootGen app.
* Browse the web page, make your selection, generate your files.

If all has gone well to this point, the framework should have generated all the source code files you need to copy into your own separate and independent Spring Boot project dedicated to your database application. But you can also do one more step by way of validation: Rebuild RADSpringBootGen with your app integrated into it, in the same manner as the FBOAce sample application is joined at the hip with RADSpringBootGen.

To build your application, whether integrated into the RADSpringBootGen framework or in your own separate and independent Spring project, the steps are very similar.

* Consult the example pom.xml and application.properties files for dependencies and configuration information. All of the dependency entries in RADSpringBootGen’s pom.xml will need to be in yours, including spring-boot-starter-thymeleaf unless you are definitively not using Thymeleaf as your top UI/HTML template layer.
* Note that the artifacts under the /static and /templates directories under src\main\resources are home to the Thymeleaf HTML templates, (cascading) stylesheets and Javascript files. Explore. The file src\main\resources\templates\layout.html is adapted from an artifact written by Marcio Marinho (of Brazil?) which pulls in the Bootstrap stylesheet framework by reference to web URLs. Depending on your firewalls and/or other security measures, you might need to download the referenced stylesheet files and build them into your project; otherwise they might fail to load at runtime and cause your UI to look very ugly indeed.
  + For more information about the Bootstrap CSS framework, see https://getbootstrap.com/docs/4.0/getting-started/introduction
  + For more information about using Bootstrap and Thymeleaf with Spring Boot in database applications, see Marcio Marinho’s tutorial “Building a CRUD Web Application with Spring Boot” at <https://www.youtube.com/watch?v=TcP5kFPq354>
* Use the directories and files under src\main\java\com\radinfodesign\RADSpringBootGen\fboace as your model for how to build your app: Substitute your app name (best in lower case) for “fboace”. If you are building your app into RADSpringBootGen, put that at the same level, under \RADSpringBootGen. If you are building your own project, it might be something like src\main\java\com\mycompany\myamazingapp. Under that, put your JPA/Hibernate-annotated model entity files in the /model directory; the repository files in the /dao directory; the service files in the /service directory and so on. The HTML edit files go under src\main\resources\templates\entity.
* Clean, compile, rebuild, deploy, lather, rinse, repeat. Note that whether you have integrated your database app into the RADSpringBootGen project or into its own separate and independent project (listening on a different port or deployed in a different (Docker?) container, you should be able to browse your modules using URLs based on your model entity names, just as they are for the sample application FBOAce. See the URL constants in the web controller template and output files.

# Part Two: Customizing the Template

## “I like it, BUT…”

Assuming the files generated against your database table and model entity classes by RADSpringBootGen perfectly suit your purposes, then you/we are done; you may close this document and enjoy happy trails using the tool more or less as-is.

But what if you want to make changes, from small tweaks to major refactoring, to how your generated files turn out? What if you have better ideas than the clown who wrote this?

For small tweaks, you need to understand how the template and its magic tokens work, which is the subject of the rest of this part. For major refactoring, you will need a detailed understanding of the framework’s architecture and internal implementation, which is the subject of [Part Three](#_Part_Three:_Enhancing).

## The Service Interface Template

On the next page(s) then we present the Service interface template, from which all of the Service interface .java source code files were generated. Have a gander:

### Service.java.template.txt

/\*

\* Generated from RADSpringBootGen

\* Copyright(c) 2018 by RADical Information Design Corporation

\* Template: ServiceTemplate.java.txt

\*/

package {${servicePackageName}$};

import {${modelPackageName}$}.{${MODEL\_ENTITY}$};

{${ACT\_THIRD\_ENTITIES\_ONLY=import {${modelPackageName}$}.{${FK\_CHILD\_ENTITY}$};

}$}import java.util.List;

public interface {${MODEL\_ENTITY}$}Service {

List<{${MODEL\_ENTITY}$}> getAll();

{${MODEL\_ENTITY}$} getEntity (Integer {${ENTITY\_ATT\_ID}$});

{${MODEL\_ENTITY}$} putEntity

( Integer {${ENTITY\_ATT\_ID}$}

{${ACT\_ALL\_ATTRIBS=, {${ENTITY\_ATTRIB\_DEFAULT\_DATATYPE}$} {${ENTITY\_ATTRIB\_NAME}$}

}$}{${ACT\_FK\_CHILD\_ENTITY\_ATTRIBS=, {${ENTITY\_ATTRIB\_DEFAULT\_DATATYPE}$}[] {${FK\_CHILD\_ENTITY\_INIT\_SMALL}$}{${ENTITY\_ATTRIB\_INITCAPS}$}s

}$}) throws Exception;

int deleteEntity(Integer {${ENTITY\_ATT\_ID}$});

!! {${ACT\_FK\_CHILD\_ENTITIES=

!! Integer delete{${FK\_CHILD\_ENTITY}$}

!! ( {${ACT\_PK\_ATTRIBS\_COMMA\_SEPARATED={${ENTITY\_ATTRIB\_DEFAULT\_DATATYPE}$} {${FK\_CHILD\_ENTITY\_INIT\_SMALL}$}{${ENTITY\_ATTRIB\_INITCAPS}$}

!! }$});

!! }$}

{${ACT\_FK\_CHILD\_ENTITIES\_W\_COMPOUND\_KEYS=int delete{${FK\_CHILD\_ENTITY}$}

( {${ACT\_PK\_ATTRIBS\_COMMA\_SEPARATED={${ENTITY\_ATTRIB\_DEFAULT\_DATATYPE}$} {${FK\_CHILD\_ENTITY\_INIT\_SMALL}$}{${ENTITY\_ATTRIB\_INITCAPS}$}

}$});

}$}{${ACT\_FK\_CHILD\_ENTITIES=int delete{${FK\_CHILD\_ENTITY}$} (Integer {${PK\_ID\_FIELD}$});

}$}

{${ACT\_THIRD\_ENTITIES\_ONLY=List<{${MODEL\_ENTITY}$}> getQualified{${MODEL\_ENTITY}$}sBy{${PK\_ID\_FIELD\_INIT\_CAP}$} (Integer {${PK\_ID\_FIELD}$});

}$}

}

### What you are seeing

Basically what you need to understand about the template is that there is:

* **Literal text**, which will be inserted verbatim into the generated output,
* **Delimited tokens:**
  + **Simple tokens** representing single-value substitution (figuring out what single value to substitute may be simple or complex), and
  + **Nested tokens**: Delimited tokens enclosing other delimited tokens.

Why single-valued and nested tokens? Because some elements to be generated, like the name of an entity, have only one value in the context of a particular entity, but the entity may have any number (many) of attributes or fields corresponding to database table columns; and may also have any number of child entities (database tables that reference the current one via foreign keys) in Collection fields; and each of those entities has attributes of their own, recursively. Child entities may also reference other entities besides the present or “driving” one (we talk a lot about driving entities), and those “Third Entities” – entities referenced by referenced entities – may be significant for the module that you are generating for your current driving entity.

For example, the Flight data model entity has the following fields:

private Integer flightId;

private String shortName;

private String longName;

private LocalDateTime departureDateTime;

private LocalDateTime arrivalDateTime;

private String notes;

private Aircraft aircraftId;

private Airport airportIdDeparture;

private Airport airportIdDestination;

private Collection<FlightCrewMember> flightCrewMemberCollection;

The complete listing of attributes as annotated is as follows:

@Id

@Basic(optional = **false**)

@Column(name = "FLIGHT\_ID")

@GeneratedValue(generator="InvSeq")

@SequenceGenerator(name="InvSeq", sequenceName="FLIGHT\_PK\_SEQ")

**private** Integer flightId;

@Basic(optional = **false**)

@Column(name = "SHORT\_NAME")

**private** String shortName;

@Column(name = "LONG\_NAME")

**private** String longName;

@Column(name = "DEPARTURE\_DATE\_TIME")

**private** LocalDateTime departureDateTime;

@Column(name = "ARRIVAL\_DATE\_TIME")

**private** LocalDateTime arrivalDateTime;

@Column(name = "NOTES")

**private** String notes;

@JoinColumn(name = "AIRCRAFT\_ID", referencedColumnName = "AIRCRAFT\_ID")

@ManyToOne(optional = **false**)

@Label (name="Aircraft")

**private** Aircraft aircraftId;

@JoinColumn(name = "AIRPORT\_ID\_DEPARTURE", referencedColumnName = "AIRPORT\_ID")

@ManyToOne

@Label (name="Departure Airport")

**private** Airport airportIdDeparture;

@JoinColumn(name = "AIRPORT\_ID\_DESTINATION", referencedColumnName = "AIRPORT\_ID")

@ManyToOne

@Label (name="Destination Airport")

**private** Airport airportIdDestination;

@OneToMany(cascade = CascadeType.***PERSIST***, mappedBy = "flight")

**private** Collection<FlightCrewMember> flightCrewMemberCollection;

These fields are represented in the putEntity() method of the Service interface by the following:

Flight putEntity

( Integer flightId

, String shortName

, String longName

, String departureDateTime

, String arrivalDateTime

, String notes

, Integer aircraftId

, Integer airportIdDeparture

, Integer airportIdDestination

, String[] flightCrewMemberNotess

, Integer[] flightCrewMemberPilotIds

) **throws** Exception;

The references from Flight to Aircraft and (two) from Flight to Airport are implemented as foreign key integer columns in the database; by object references in the entity class; and again as integers in the Services and Web Controllers. This conversion of one representation to another and back again is one of the recurring challenges of any Java-based relational database application.

The last two arguments String[] flightCrewMemberNotess (the 2 s’s are NOT a typo) and Integer[] flightCrewMemberPilotIds, represent the collection field

**private** Collection<FlightCrewMember> flightCrewMemberCollection;

…rendered as arrays of attribute values. The FLIGHT\_CREW\_MEMBER table is an associative entity, a.k.a. a many-to-many resolver between the FLIGHT and PILOT tables. A Flight may have one or more Pilots associated with it as crew members, and each Pilot may be associated with zero, one or more Flights. This classic many-to-many relationship is represented and resolved by the FLIGHT\_CREW\_MEMBER table and its corresponding FlightCrewMember model entity.

***Digression: Some Entities are more primary than others***

*While it is possible to build a complete stack of model entities, repositories, service interfaces, service implementation classes, controllers and data maintenance forms for every single table in the database schema, in practice associative entities like FlightCrewMember and PilotCertification (which cross-references Pilots with AircraftType to indicate the type of aircraft they are qualified to fly) which have no children of their own, do not require services, controllers and UI form of their own. It is enough to incorporate their elements into the modules of their parent entities.*

Back to our template, its literal text, delimited tokens and nested tokens: The Flight putEntity() method listed above (and all the putEntity() methods for the other entities) was generated from the following template text mixed with the (Flight) entity metadata.

{${MODEL\_ENTITY}$} putEntity

( Integer {${ENTITY\_ATT\_ID}$}

{${ACT\_ALL\_ATTRIBS=, {${ENTITY\_ATTRIB\_DEFAULT\_DATATYPE}$} {${ENTITY\_ATTRIB\_NAME}$}

}$}{${ACT\_FK\_CHILD\_ENTITY\_ATTRIBS=, {${ENTITY\_ATTRIB\_DEFAULT\_DATATYPE}$}[] {${FK\_CHILD\_ENTITY\_INIT\_SMALL}$}{${ENTITY\_ATTRIB\_INITCAPS}$}s

}$}) throws Exception;

Confused yet? Let’s back up and walk through how your template is processed, assuming we know the structure of your database tables, the source code of the model entity and repository classes, and the content of your template file, using Airport as the entity and Service.java.template.txt as our entry-level examples.

Here are the first 8 lines of the template:

1 /\*

2 \* Generated from RADSpringBootGen

3 \* Copyright(c) 2018 by RADical Information Design Corporation

4 \* Template: ServiceTemplate.java.txt

5 \*/

6 package {${servicePackageName}$};

7

8 import {${modelPackageName}$}.{${MODEL\_ENTITY}$};

Everything (line numbers excluded) from the first “/\*” until the space after the keyword “package” on line 6 is literal text that will be inserted into the output exactly as it is read from the input. Then there is the simple, single-valued token “servicePackageName” wrapped with the RADSpringBootGen delimiters “{${” and “}$}”. Then there is a semicolon followed by two carriage returns and linefeeds, followed by the word “import” and a space, all of which is interpreted as literal text. Then there is another delimited token, followed by a literal period, followed by another delimited token, followed by a literal semicolon.

And all of that results in the output (translated delimited tokens underlined):

/\*

\* Generated from RADSpringBootGen

\* Copyright(c) 2018 by RADical Information Design Corporation

\* Template: ServiceTemplate.java.txt

\*/

package com.radinfodesign.RADSpringBootGen.fboace.service;

import com.radinfodesign.RADSpringBootGen.fboace.model.Airport;

Make sense? {${servicePackageName}$} just means “insert the Service Package Name, which, based on the user’s input and confirmed by reading classes and source code files, is ‘com.radinfodesign.RADSpringBootGen.fboace.service’, here.”

{${modelPackageName}$} similarly resolves to the corresponding model package name. And {${MODEL\_ENTITY}$} resolves to “Airport”. wherever that token may appear in the template for as long as we are processing Airport as the primary, driving entity.

The service and model package names are the same for all entities within a generated application, based on the base application package. But {${MODEL\_ENTITY}$} is of course different in each case: Airport, AircraftType, Aircraft, Pilot, Flight etc.

Let’s skip a couple of lines in the template that are not applicable (and therefore will be skipped by the code generator) for a simple entity like Airport, and look at a few more lines:

public interface {${MODEL\_ENTITY}$}Service {

List<{${MODEL\_ENTITY}$}> getAll();

{${MODEL\_ENTITY}$} getEntity (Integer {${ENTITY\_ATT\_ID}$});

{${MODEL\_ENTITY}$} putEntity

( Integer {${ENTITY\_ATT\_ID}$}

Can you read and interpret this, predict what its “translated” output will be? We’ve already seen {${MODEL\_ENTITY}$}, which we may re-use wherever we need the name of the model entity (including the capitalized initial letter). The only new element here is the token {${ENTITY\_ATT\_ID}$}. Can you infer what it means?

That’s right! It’s the name of the primary key identifier field for the entity, in the present case, “airportId”. For Pilot it would be “pilotId”, for Aircraft, “aircraftId”; for Flight, “flightId”. Or you could name your single-column primary key ID field “id” for all of your entities, even if that’s different from the database table primary key column name. Whatever suits you; whatever it happens to be.

And so, those few lines of the template above result in the following when generated against the Airport.java definition:

public interface AirportService {

List<Airport> getAll();

Airport getEntity (Integer airportId);

Airport putEntity

( Integer airportId

### Nested/Multivalued tokens: putEntity()

Here’s where it gets interesting. Most data model entities have one and only one ID field (primary key column), but they may have any number of other simple attributes, other-entity (foreign key) references or child entity collections.

The generically named putEntity() method serves to allow the web controller class to ‘put’ all of the values of the attributes of a new or already existing entity, so that a new one may be inserted or an existing one may be updated (it also provides for the insertion and/or updating of one or more child entity instances; in this way an instance of an “entity” is treated more as a comprehensive, rich *object* and not just a single row in a single database table).

The complete signatures for the putEntity() methods for the Airport, Flight and Pilot entities are as follows:

**AirportService.java**

Airport putEntity

( Integer airportId

, String shortName

, String iataCode

, String description

, String portType

) throws Exception;

**FlightService.java**

Flight putEntity

( Integer flightId

, String shortName

, String longName

, String departureDateTime

, String arrivalDateTime

, String notes

, Integer aircraftId

, Integer airportIdDeparture

, Integer airportIdDestination

, String[] flightCrewMemberNotess

, Integer[] flightCrewMemberPilotIds

) throws Exception;

**PilotService.java**

Pilot putEntity

( Integer pilotId

, String lastName

, String firstName

, String middleInitial

, String nationalIdNumber

, String birthdate

, String notes

, String[] pilotCertificationCertificationNumbers

, String[] pilotCertificationValidFromDates

, String[] pilotCertificationExpirationDates

, String[] pilotCertificationNotess

, Integer[] pilotCertificationAircraftTypeIds

, String[] flightCrewMemberNotess

, Integer[] flightCrewMemberFlightIds

) throws Exception;

In spite of the wide variability among these code segments, they are all generated from the same template; it is the definition of each of the entities that accounts for the variations.

Look at the template snippet again:

{${MODEL\_ENTITY}$} putEntity

( Integer {${ENTITY\_ATT\_ID}$}

{${ACT\_ALL\_ATTRIBS=, {${ENTITY\_ATTRIB\_DEFAULT\_DATATYPE}$} {${ENTITY\_ATTRIB\_NAME}$}

}$}{${ACT\_FK\_CHILD\_ENTITY\_ATTRIBS=, {${ENTITY\_ATTRIB\_DEFAULT\_DATATYPE}$}[] {${FK\_CHILD\_ENTITY\_INIT\_SMALL}$}{${ENTITY\_ATTRIB\_INITCAPS}$}s

}$}) throws Exception;

We saw the first two lines explained earlier. The next significant unit is:

{${ACT\_ALL\_ATTRIBS=, {${ENTITY\_ATTRIB\_DEFAULT\_DATATYPE}$} {${ENTITY\_ATTRIB\_NAME}$}

}$}

ACT\_ALL\_ATTRIBS is a nested token; ENTITY\_ATTRIB\_DEFAULT\_DATATYPE and ENTITY\_ATTRIB\_NAME are simple, single-valued tokens within the outer nesting. ACT\_ALL\_ATTRIBS instructs the generator to ACT upon this instruction for ALL of the (simple, non-key and non-collection) attributes of the present entity. In the case of Airport, that means shortName, iataCode, description and portType. ENTITY\_ATTRIB\_DEFAULT\_DATATYPE simply means “insert the datatype of the entity attribute here”.

And so with that we get (for Airport):

, String shortName

, String iataCode

, String description

, String portType

Note that the spaces before the opening delimiters of the nested token

\_ \_ \_ \_ \_ \_{${ACT\_ALL\_ATTRIBS=

…as well as the comma and space following that, are treated as literal text to be inserted into the output. No element, no byte of the template is without significance or consequence.

So far so good. But how did we get

**FlightService.java**

, String[] flightCrewMemberNotess

, Integer[] flightCrewMemberPilotIds

**PilotService.java**

, String[] pilotCertificationCertificationNumbers

, String[] pilotCertificationValidFromDates

, String[] pilotCertificationExpirationDates

, String[] pilotCertificationNotess

, Integer[] pilotCertificationAircraftTypeIds

, String[] flightCrewMemberNotess

, Integer[] flightCrewMemberFlightIds

???

Those lines result from the template segment

{${ACT\_FK\_CHILD\_ENTITY\_ATTRIBS=, {${ENTITY\_ATTRIB\_DEFAULT\_DATATYPE}$}[] {${FK\_CHILD\_ENTITY\_INIT\_SMALL}$}{${ENTITY\_ATTRIB\_INITCAPS}$}s

}$}

ACT\_FK\_CHILD\_ENTITY\_ATTRIBS means, for each child entity of the current driving entity (in the case of flight, FlightCrewMember; in the case of Pilot, PilotCertification and FlightCrewMember), ACT upon all of their attributes, excluding the one that references back to the driving entity. Accordingly, for FlightService, an array of flightCrewMemberPilotIds is included while flightCrewMemberFlightIds is not, and for Pilot it is the other way around; we already know the present parent key.

We saw ENTITY\_ATTRIB\_DEFAULT\_DATATYPE explained previously. FK\_CHILD\_ENTITY\_INIT\_SMALL returns the name of the child entity (which references the current driving/parent entity via a foreign key, hence “FK”) with an initial lower-case letter instead of the upper case of a typical class name. And ENTITY\_ATTRIB\_INITCAPS returns the name of each attribute of the entity currently being processed (and because this occurrence is nested inside of ACT\_FK\_CHILD\_ENTITY\_ATTRIBS, the

You may have guessed, if an entity doesn’t have any child entities, then the instruction to act upon the latter is simply ignored without error or other consequence, as in the case of Airport.

Please review our discussion of how the putEntity() method is generated and make sure you have mastered the concept. This is the simplest example of the use of nested tokens in the template to generate potentially multivalued attributes in the output, and serves as the foundation for all subsequent discussions.

### deleteEntity()

The signature for the generic deleteEntity() method is very simple, and similar for almost all entities:

**AirportService.java**

int deleteEntity(Integer airportId);

**FlightService.java**

int deleteEntity(Integer flightId);

**PilotService.java**

int deleteEntity(Integer pilotId);

… and all generated from the template line (which should be be obvious to you by now):

int deleteEntity(Integer {${ENTITY\_ATT\_ID}$});

### Template Comments

The next lines in the Service.java.template.txt introduce another concept:

!! {${ACT\_FK\_CHILD\_ENTITIES=

!! Integer delete{${FK\_CHILD\_ENTITY}$}

!! ( {${ACT\_PK\_ATTRIBS\_COMMA\_SEPARATED={${ENTITY\_ATTRIB\_DEFAULT\_DATATYPE}$} {${FK\_CHILD\_ENTITY\_INIT\_SMALL}$}{${ENTITY\_ATTRIB\_INITCAPS}$}

!! }$});

!! }$}

Whatever these lines in the template *might* otherwise do, as it happens, *they do nothing at all*, because the symbols !! at the beginning of a line signify comments in the framework; what follows in any line that begins with double exclamation marks is ignored for purposes of code generation (watch for wrapped lines).

### Delete Child Instances

The FlightService and PilotService interfaces provide facilities for deleting child entity instances:

**FlightService.java**

int deleteFlightCrewMember

( Integer flightCrewMemberFlightId

, Integer flightCrewMemberPilotId

);

**PilotService.java**

int deletePilotCertification

( Integer pilotCertificationPilotId

, Integer pilotCertificationAircraftTypeId

);

int deleteFlightCrewMember

( Integer flightCrewMemberFlightId

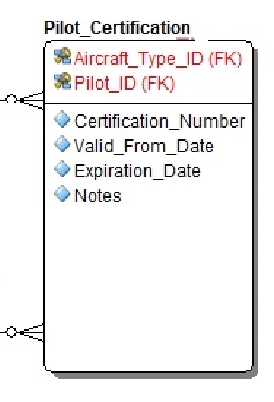
, Integer flightCrewMemberPilotId

);

As it happens, to identify and instance/table row to delete in the case of FlightCrewMember and PilotCertification requires two elements, not just one.

Recall the data model for these associate entities:





The primary key of each of these is comprised of the key of both parent entities. Instead of a simple Integer attribute the keys then are represented by the classes FlightCrewMemberPK and PilotCertificationPK, which encapsulate the multiple attributes.

The above code snippets are generated from the following template lines (some wrapped awkwardly here):

{${ACT\_FK\_CHILD\_ENTITIES\_W\_COMPOUND\_KEYS=int delete{${FK\_CHILD\_ENTITY}$}

( {${ACT\_PK\_ATTRIBS\_COMMA\_SEPARATED={${ENTITY\_ATTRIB\_DEFAULT\_DATATYPE}$} {${FK\_CHILD\_ENTITY\_INIT\_SMALL}$}{${ENTITY\_ATTRIB\_INITCAPS}$}

}$});

}$}

Can you decipher it yet? ACT\_FK\_CHILD\_ENTITIES\_W\_COMPOUND\_KEYS means “For each child entity that has compound primary keys, act: process each of the key elements.” FK\_CHILD\_ENTITY returns the name of each child entity.

ACT\_PK\_ATTRIBS\_COMMA\_SEPARATED returns the primary key attributes as a comma-separated list. Note that this token is nested within ACT\_FK\_CHILD\_ENTITIES\_W\_COMPOUND\_KEYS, and it also nests other tokens within itself: multilevel nesting. VERY IMPORTANT: Count and note the positions of the opening and closing delimiters. If you get this wrong, just like nested curly-brace blocks of Java code, your results will not be what you intend.

The other tokens we have seen before, and they behave exactly the same way as before.

*To be explained in a future version of this document, time permitting:*

*{${ACT\_FK\_CHILD\_ENTITIES=int delete{${FK\_CHILD\_ENTITY}$} (Integer {${PK\_ID\_FIELD}$});*

*}$}*

*{${ACT\_THIRD\_ENTITIES\_ONLY=List<{${MODEL\_ENTITY}$}> getQualified{${MODEL\_ENTITY}$}sBy{${PK\_ID\_FIELD\_INIT\_CAP}$} (Integer {${PK\_ID\_FIELD}$});*

*}$}*

*}*

## Token, token, Who’s got the Token?

That covers the key concepts of the RADSpringBootGen templating system, using a combination of literal text, simple delimited tokens and nested tokens to achieve an output based on a data model entity’s attributes and relationships with other entities. In a future revision, we may develop these step-by-step explanations in greater detail, to even greater degrees of complexity. For now, chew on this: the comprehensive catalog of token values and their implicit meaning/results that RADSpringBootGen offers to you, from the class **OutputStringTree**.

**public** **class** OutputStringTree **extends** IOStringTree {

// public class OutputStringTreeImpl extends IOStringTree implements OutputStringTree {

// Constants named "ACT\_\*" indicate instruction to ACT upon various elements of table/EntityMeta or column/attribute/FieldMeta

**final** String ACT\_ALL\_ATTRIBS = "ACT\_ALL\_ATTRIBS"; // Flag indicating logic applying to all (non-collection) attributes of the entity

**final** String ACT\_UI\_ATTRIBS = "ACT\_UI\_ATTRIBS"; //

**final** String ACT\_SIMPLE\_ATTRIBS = "ACT\_SIMPLE\_ATTRIBS"; // Flag indicating logic applying only to simple attributes (non-temporal primitives and wrapper types)

**final** String ACT\_DATE\_ATTRIBS = "ACT\_DATE\_ATTRIBS"; // Flag indicating logic applying only to Date (LocalDate) type attributes

**final** String ACT\_DATE\_TIME\_ATTRIBS = "ACT\_DATE\_TIME\_ATTRIBS"; // Flag indicating logic applying only to Datetime (LocalDateTime) type attributes

**final** String ACT\_FK\_REF\_ENTITIES = "ACT\_FK\_REF\_ENTITIES"; // Flag indicating logic applying only to FK-referenced entities, elimintating duplicate/redundant entries for the same referenced entity class

**final** String ACT\_NON\_TEMPORAL\_ATTRIBS = "ACT\_NON\_TEMPORAL\_ATTRIBS"; // Flag indicating logic applying to non-temporal (LocalDate or LocalDateTimeTime) attributes

**final** String ACT\_FK\_REF\_ATTRIBS = "ACT\_FK\_REF\_ATTRIBS"; // Flag indicating logic applying only to FK-referenced entities; no duplicate elimination

**final** String ACT\_NON\_KEY\_ATTRIBS = "ACT\_NON\_KEY\_ATTRIBS"; // Flag indicating logic applying only to attributes that are NOT members of the primary key

**final** String ACT\_PK\_ATTRIBS = "ACT\_PK\_ATTRIBS"; // Flag indicating logic applying only to attributes that ARE members of the primary key

**final** String ACT\_PK\_ATTRIBS\_COMMA\_SEPARATED = "ACT\_PK\_ATTRIBS\_COMMA\_SEPARATED"; // Flag indicating logic applying only to attributes that ARE members of the primary key, separated by commas in the case of compound key

**final** String ACT\_FK\_CHILD\_ENTITIES = "ACT\_FK\_CHILD\_ENTITIES"; // Flag indicating logic applying only to child entities.

**final** String ACT\_FK\_CHILD\_ENTITIES\_W\_COMPOUND\_KEYS = "ACT\_FK\_CHILD\_ENTITIES\_W\_COMPOUND\_KEYS"; // Flag indicating logic applying only to child entities have compound primary keys, one component being inherited from the Driving Entity

**final** String ACT\_FK\_CHILD\_ENTITIES\_W\_COMPOUND\_KEYS\_FORCE\_INCLUDE = "ACT\_FK\_CHILD\_ENTITIES\_W\_COMPOUND\_KEYS\_FORCE\_INCLUDE"; // Include this node even if FieldMeta.isExcludedEditFromParentModule()

**final** String ACT\_FK\_CHILD\_ENTITIES\_FORCE\_INCLUDE = "ACT\_FK\_CHILD\_ENTITIES\_FORCE\_INCLUDE"; // Include this node even if FieldMeta.isExcludedEditFromParentModule()

**final** String ACT\_FK\_CHILD\_AND\_THIRD\_ENTITIES = "ACT\_FK\_CHILD\_AND\_THIRD\_ENTITIES"; // Flag indicating logic applying only to child entities and entities referenced by child entities, excluding the driving entity.

**final** String ACT\_FK\_CHILD\_W\_COMPOUND\_KEYS\_AND\_THIRD\_ENTITIES = "ACT\_FK\_CHILD\_W\_COMPOUND\_KEYS\_AND\_THIRD\_ENTITIES"; // Logic applying only to child entities with compound primary keys, plus any third entities referenced by the same.

**final** String ACT\_THIRD\_ENTITIES\_ONLY = "ACT\_THIRD\_ENTITIES\_ONLY"; // Flag indicating logic applying only to third entities referenced by child entities that have compound primary keys.

// final String ACT\_IF\_THIRD\_ENTITIES\_EXIST = "ACT\_IF\_THIRD\_ENTITIES\_EXIST"; // Process the enclosed nodes only if the current driving entity is related to third entities through child/associative entities

**final** String ACT\_OTHER\_REF\_ENTITIES = "ACT\_OTHER\_REF\_ENTITIES";

**final** String ACT\_FK\_CHILD\_ENTITY\_ATTRIBS = "ACT\_FK\_CHILD\_ENTITY\_ATTRIBS"; // Flag indicating logic applying to attributes of child entities, excluding those referencing the driving entity (for now; how to handle multiple references? Future bug?)

**final** String ACT\_FK\_CHILD\_ENTITY\_W\_COMPOUND\_KEY\_ATTRIBS = "ACT\_FK\_CHILD\_ENTITY\_W\_COMPOUND\_KEY\_ATTRIBS"; // Flag indicating logic applying to attributes of child entities that have compound primary keys (presumably inheriting one member from the present driving entity.

**final** String ACT\_THIRD\_ENTITIES = "ACT\_THIRD\_ENTITIES"; // Flag indicating logic applying only to other entities referenced by child entities. Refinement of ACT\_FK\_CHILD\_ENTITIES

**final** String ACT\_FK\_CHILD\_EMBEDDED\_ID = "ACT\_FK\_CHILD\_EMBEDDED\_ID"; // Flag indicating logic applying only to the embedded ID attribute of child entities.

**final** String ACT\_ALL\_FK\_CHILD\_ENTITIES = "ACT\_ALL\_FK\_CHILD\_ENTITIES"; // New/revised Flag indicating logic applying to child entities, whether having simple or compound primary keys

**final** String ACT\_ALL\_FK\_CHILD\_ENTITIES\_FORCE\_INCLUDE = "ACT\_ALL\_FK\_CHILD\_ENTITIES\_FORCE\_INCLUDE"; // New/revised Flag indicating logic applying to child entities, whether having simple or compound primary keys

**final** String FORCE\_INCLUDE = "FORCE\_INCLUDE";

**final** String FK\_REF\_ENTITY = "FK\_REF\_ENTITY"; // Name of Entity class referenced by foreign key

**final** String FK\_REF\_ENTITY\_QUALIFIED = "FK\_REF\_ENTITY\_QUALIFIED"; // Qualified identifier of Entity class referenced by foreign key

**final** String FK\_REF\_ENTITY\_ID = "FK\_REF\_ENTITY\_ID"; // Primary key ID field of Entity class referenced by foreign key

**final** String FK\_REF\_ENTITY\_ID\_INIT\_CAP = "FK\_REF\_ENTITY\_ID\_INIT\_CAP"; // Primary key ID field of Entity class referenced by foreign key

**final** String FK\_REF\_ENTITY\_INIT\_SMALL = "FK\_REF\_ENTITY\_INIT\_SMALL"; // Name of Entity class referenced by foreign key

**final** String FK\_REF\_ENTITY\_LOWER = "FK\_REF\_ENTITY\_LOWER"; // Name of child Entity class in lowercasse

**final** String FK\_CHILD\_ENTITY = "FK\_CHILD\_ENTITY"; // Name of Entity class that is a child of the primary

**final** String FK\_CHILD\_ENTITY\_IDENTIFIER = "FK\_CHILD\_ENTITY\_IDENTIFIER"; // Name given to reference to collection of Entity class that is a child of the primary, in context of parent

**final** String FK\_CHILD\_ENTITY\_QUALIFIED = "FK\_CHILD\_ENTITY\_QUALIFIED"; // Qualified identifier of Entity class that is a child of the primary

**final** String FK\_CHILD\_ENTITY\_INIT\_SMALL = "FK\_CHILD\_ENTITY\_INIT\_SMALL"; // Name of child Entity class referenced by foreign key

**final** String FK\_CHILD\_ENTITY\_LOWER = "FK\_CHILD\_ENTITY\_LOWER"; // Name of Entity class referenced by foreign key, in lowercase

**final** String FK\_CHILD\_ENTITY\_UPPER = "FK\_CHILD\_ENTITY\_UPPER"; // Name of Entity class referenced by foreign key, in UPPERCASE

**final** String FK\_CHILD\_ENTITY\_LOWER\_PLURAL = "FK\_CHILD\_ENTITY\_LOWER\_PLURAL"; // Lowercase plural name of child entity class

**final** String FK\_CHILD\_ENTITY\_UPPER\_PLURAL = "FK\_CHILD\_ENTITY\_UPPER\_PLURAL"; // UPPERCASE plural name of child entity class

**final** String FK\_CHILD\_ENTITY\_LABEL = "FK\_CHILD\_ENTITY\_LABEL";

**final** String FK\_CHILD\_TO\_REF\_ENTITY\_VAR\_EXPR = "FK\_CHILD\_TO\_REF\_ENTITY\_VAR\_EXPR"; //

**final** String FK\_CHILD\_EMBEDDED\_ID = "FK\_CHILD\_EMBEDDED\_ID"; // Embedded ID (foreign key column) field.

**final** String FK\_CHILD\_EMBEDDED\_ID\_INIT\_CAPS = "FK\_CHILD\_EMBEDDED\_ID\_INIT\_CAPS"; // Embedded ID (foreign key column) field in initial caps.

**final** String FK\_CHILD\_EMBEDDED\_ID\_INIT\_SMALL = "FK\_CHILD\_EMBEDDED\_ID\_INIT\_SMALL"; // Embedded ID (foreign key column) field in initial lower case.

**final** String FK\_CHILD\_EMBEDDED\_PK = "FK\_CHILD\_EMBEDDED\_PK"; // Child Entity Embedded PK object.

**final** String FK\_CHILD\_EMBEDDED\_PK\_INIT\_SMALL = "FK\_CHILD\_EMBEDDED\_PK\_INIT\_SMALL"; // Child Entity Embedded PK object in initial lowercase.

**final** String PK\_ID\_FIELD = "PK\_ID\_FIELD"; // Name of single primary key/ID field

**final** String PK\_ID\_FIELD\_INIT\_CAP = "PK\_ID\_FIELD\_INIT\_CAP"; // Name of single primary key/ID field

**final** String PK\_FK\_REF\_ENTITY = "PK\_FK\_REF\_ENTITY"; // Name of entity referenced by foreign key and (embedded) primary key

**final** String PK\_FK\_REF\_ENTITY\_INIT\_SMALL = "PK\_FK\_REF\_ENTITY\_INIT\_SMALL"; // ...with initial lowercase letter

**final** String PK\_FK\_REF\_ENTITIES\_DECLARE\_REPOSITORY\_FIND = "PK\_FK\_REF\_ENTITIES\_DECLARE\_REPOSITORY\_FIND"; // Complete multiple delarations and initializations of Entities referenced by embedded PK object by call to repository.findOne()

// Example: Pilot pilot = pilotRepository.findOne(flightCrewMemberPilotId);

**final** String FIND\_ONE\_BY\_PK\_FK\_CRITERIA = "FIND\_ONE\_BY\_PK\_FK\_CRITERIA";

**final** String CALL\_COMPOUND\_CONSTRUCTOR = "CALL\_COMPOUND\_CONSTRUCTOR";

**final** String GET\_TH\_HTML\_FORM\_DATA\_VARS = "GET\_TH\_HTML\_FORM\_DATA\_VARS";

**final** String COMPOUND\_PK\_PARAM\_LIST = "COMPOUND\_PK\_PARAM\_LIST";

**final** String COMPOUND\_PK\_PARAM\_LIST\_CHILD\_ENTITY = "COMPOUND\_PK\_PARAM\_LIST\_CHILD\_ENTITY";

**final** String COMPOUND\_INSERT\_PARAM\_LIST\_CHILD\_ENTITY = "COMPOUND\_INSERT\_PARAM\_LIST\_CHILD\_ENTITY";

**final** String FIRST\_NON\_KEY\_REQUIRED\_ATTRIB = "FIRST\_NON\_KEY\_REQUIRED\_ATTRIB"; // Name of the first required non-key field

**final** String FIRST\_NON\_KEY\_REQUIRED\_ATTRIB\_INIT\_CAP = "FIRST\_NON\_KEY\_REQUIRED\_ATTRIB\_INIT\_CAP"; // Name of the first required non-key field, with initial capital letter

**final** String FK\_REF\_ATTRIB\_NAME = "FK\_REF\_ATTRIB\_NAME"; // Name of Entity member attribute representing class referenced by foreign key

**final** String FK\_REF\_ATTRIB\_INITCAPS = "FK\_REF\_ATTRIB\_INITCAPS"; // Name of Entity member attribute representing class referenced by foreign key, with Initial Capital

**final** String FK\_REF\_ENTITY\_LOWER\_PLURAL = "FK\_REF\_ENTITY\_LOWER\_PLURAL"; // Lowercase plural name of entity class referenced by foreign key, suitable as reference variable name

**final** String FK\_REF\_ENTITY\_UPPER\_PLURAL = "FK\_REF\_ENTITY\_UPPER\_PLURAL"; // UPPERCASE plural name of entity class referenced by foreign key, suitable as reference variable name

**final** String ENTITY\_ATTRIB\_UPPER\_NAME = "ENTITY\_ATTRIB\_UPPER\_NAME"; // Upper-case underscore-separated name of entity attribute (table column)

**final** String ENTITY\_ATTRIB\_NAME = "ENTITY\_ATTRIB\_NAME"; // Entity attribute name

**final** String ENTITY\_ATTRIB\_LABEL = "ENTITY\_ATTRIB\_LABEL"; // Entity attribute formatted initcaps label

**final** String ENTITY\_ATTRIB\_DEFAULT\_DATATYPE = "ENTITY\_ATTRIB\_DEFAULT\_DATATYPE"; // Entity attribute datatype; Non-primitive non-wrappers (entity classes) return Integer; Dates return String

**final** String ENTITY\_ATTRIB\_INITCAPS = "ENTITY\_ATTRIB\_INITCAPS"; // Entity attribute name with initial capital, suitable in construction of prefixed identifier

**final** String ENTITY\_DATE\_ATTRIB\_NAME = "ENTITY\_DATE\_ATTRIB\_NAME"; // Date type attribute name

**final** String ENTITY\_DATE\_TIME\_ATTRIB\_NAME = "ENTITY\_DATE\_TIME\_ATTRIB\_NAME"; // Datetime type attribute name

**final** String ENTITY\_DATE\_ATTRIB\_INITCAPS = "ENTITY\_DATE\_ATTRIB\_INITCAPS"; // Date type attribute name

**final** String ENTITY\_DATE\_TIME\_ATTRIB\_INITCAPS = "ENTITY\_DATE\_TIME\_ATTRIB\_INITCAPS"; // Datetime type attribute name

**final** String ENTITY\_ATRRIB\_INITCAP\_NAME = "ENTITY\_ATRRIB\_INITCAP\_NAME"; // Entity attribute name with InitialCapital

**final** String MODEL\_ADD\_CHILD\_ENTITY\_RPSTRY\_ATTRIB = "MODEL\_ADD\_CHILD\_ENTITY\_RPSTRY\_ATTRIB";

// final String MODEL\_ENTITY\_IMPORT = "modelEntityImport"; // Token to retrieve fully-qualified name of primary driving entity class

**final** String HTML\_FORM\_VERTICAL\_INPUT = "HTML\_FORM\_VERTICAL\_INPUT";

**final** String HTML\_FORM\_HORIZONTAL\_INPUT = "HTML\_FORM\_HORIZONTAL\_INPUT";

**final** String HTML\_FORM\_VERTICAL\_INPUT\_BLANK = "HTML\_FORM\_VERTICAL\_INPUT\_BLANK";

**final** String HTML\_FORM\_HORIZONTAL\_INPUT\_BLANK = "HTML\_FORM\_HORIZONTAL\_INPUT\_BLANK";

W

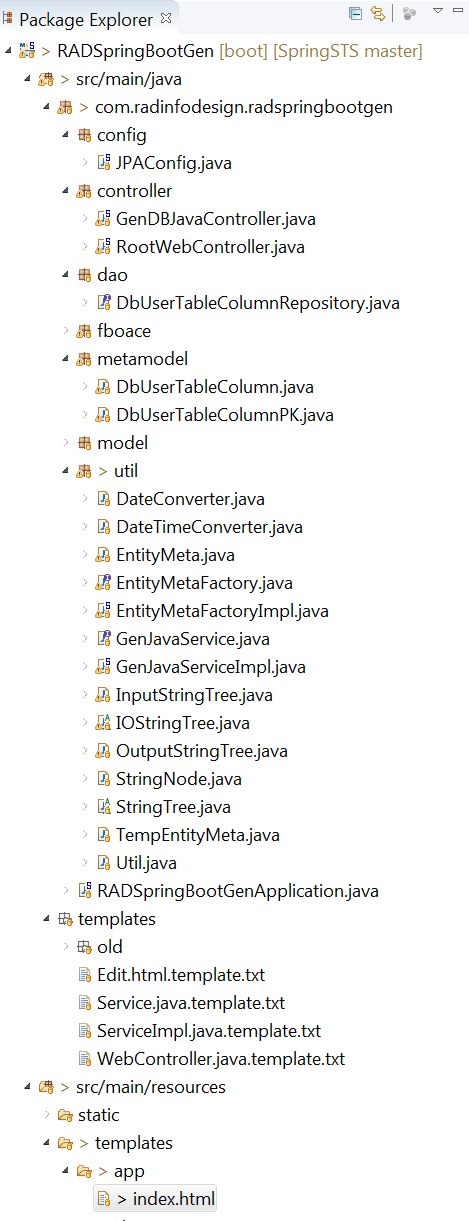
# Part Three: Enhancing RADSpringBootGen’s Functionality

## Internal Technical Documentation

Part One taught you what RADSpringBootGen is and how to use it out-of-the-box to generate artifacts for your database application. Part Two taught you how to customize your templates to your exact requirements, by explaining the structure and function of literal text, simple delimited tokens and complex, nested tokens; and the library of token values available for your use, and what to expect from each when you use it. In this part, we take customization a step further, enabling you to extend the functionality of RADSpringBootGen by giving you a window into the internal implementation of the framework’s Java classes; enabling radical refactoring.

Welcome to the inner sanctum. Along the way, you may learn some concepts and techniques that will make you a star programmer.

Here is a screenshot of the Package Explorer pane of SpringSTS for the RADSpringBootGen project, with the most relevant classes and files showing:



### Launch!

Here is what happens when the RADSpringBootGen web app user (you) clicks on the [Launch Code Generation] button, submitting the HTML form.

* app/index.html form invokes JavaController.launchGen()
* genJavaService.launch()
  + Nested loops on entity list and components to be generated (Service interface, ServiceImpl class, Web Controller, Edit.html)
* genJavaService.genCodeFromTemplate ()
  + InputStringTree parses template into tree of StringNodes
  + OutputStringTree processes InputStringTree with model entity metadata: OutputStringTree.**build()**
    - Tests each node for isLiteralExpression(), isSingleTokenExpression(), isMultiTokenExpression(), processes accordingly.
    - Recursively calls build() to processes all child nodes of current node
    - (Top node is the whole template file and as such fails all three above tests; so first pass falls through to recursive processing of child nodes.)

# [TO BE CONTINUED…]

1. “About which more anon” = a fancy British expression meaning “I’ll tell you more about that later.” [↑](#footnote-ref-1)