



**Capstone Courseware, LLC**

33 Boylston Street  
Jamaica Plain, MA 02130

877-227-2477  
capstonecourseware.com

# The Spring Framework

**Version 2.0.1**

## **Instructor's Guide**



## Overview

Let's begin this guide for instructors more or less the way we begin the coursebook: by taking cognizance of the sheer size of the Spring framework. Certainly, no four-day course is going to do justice to the full scope and depth of this code library. This is by no means a survey course, and the boundaries around the course scope are more in the dimension of breadth than depth. But the boundaries are nevertheless there: we can't delve into every detail of the framework.

Also, with Spring as with any open-source tool, we're working with the brilliant ideas of a few bright people; we are not teaching an elected standard such as Servlets or JSP. And with a project as popular and dynamic as Spring, that step apart from standards-based frameworks is not the last one; there is a lively aftermarket in Spring utilities, helpers, IDEs, etc.

All of this suggests that it is better to teach this course from a position of enthusiasm for what the tool can do, than by projecting a sense of explaining exactly how it's supposed to be done. That is, if in Servlets training we say, "thou shalt extend HttpServlet," in Spring we tend to say, "here's one way to get your controllers mapped neatly to your requests." For every way of doing something in Spring that this course can show, there will be several others, and some of them will involve code not found in the Spring JARs. Spring is in a way the glue between the well-considered but somewhat static Java EE platform and the best practices and inspirations of any experienced and dedicated Java developer. The most responsive audiences will be those with some real Java EE experience under their belts – those who've done it the hard way a few times and will appreciate the subtle advantages of Spring's inversion of control and general respect for design patterns. Novice audiences – I'm thinking of the students who technically have the prerequisites but maybe not such deep experience – will probably need a little more in the way of evangelism: don't hesitate to get up on that soapbox if it seems necessary, because the details will be a lot more interesting to students who buy the argument in the first place that IoC is good, strategies are good, MVC is good, etc.

This course has evolved from Course 117 – same title, much of the same code, but a day longer and with more material to create a less steep learning curve, especially on the Core and Web modules. We hope that instructors will find this updated and expanded course to make for a smoother, less jarring learning experience – especially for those relatively inexperienced Java developers who tend to appear in class, despite our stated prerequisites, but for more senior people as well. See the revision history for details of the transition from 117 to 117A.



## Timeline

The following breakdowns are approximate, and every class will vary.

### Day 1

2 hours	Chapter 1
3½ hours	Chapter 2
1 hour, runs to day 2	Chapter 3

### Day 2

3½ hours	Chapter 3
1½ hours	Chapter 4
1½ hours, runs to day 3	Chapter 5

### Day 3

2 hours	Chapter 5
1 hour	Chapter 6
2½ hours	Chapter 7
1 hour, runs to day 4	Chapter 8

### Day 4

½ hour	Chapter 8
2½ hours	Chapter 9
1½ hours	Chapter 10
2 hours	Chapter 11

If time is tight, the best candidates for skipping or skimming are probably Chapters 6 and 10.





## Crimson and the XML Validator

For those classrooms not using Eclipse, it will be worth the trouble to introduce students to a simple XML authoring environment deployed with the course software. The setup guide requires that the Crimson text editor be installed to `c:\Capstone\Tools\Crimson3.70`; then, the lab installer bundles an XML validating tool from Capstone Courseware (just a command-line interface to Xerces, really) and a single file that installs custom hotkeys into Crimson. The end result is that Crimson acts as an integrated editor, parser and schema validator for XML files. Hit **F9** to parse for well-formedness and **F10** to validate against a schema that's referenced in the XML document using the **xsi:schemaLocation** attribute.

Students may find this helpful when creating and editing their beans XML files and web context configurations. Especially if you sense that students' XML experience is slight, take a few minutes to show them this environment, and suggest that they make a habit of saving with **F10** rather than simply **File|Save**; this will trigger schema validation of their XML files and alert them to any of the common syntax or validity mistakes that can otherwise become frustrating as they don't surface until after the whole application's been run or deployed to Tomcat.

Two unfortunate limitations: one is that, on the one hand, Spring's bean factories insist on a **schemaLocation** attribute, or they fail to load the XML. This is non-compliant with XML Schema, but fairly harmless as we prime all our files with this attribute. On the other hand, web application contexts will choke if the **schemaLocation** is there and can't be resolved at runtime! So for safety's sake we leave them out of the context configurations in all the web projects. This in turn means that **F10** won't be useful for these files; students can still hit **F9** to check basic syntax. If one suspects a schema-validity problem but can't quite track it down by eye, one can always add the **schemaLocation** phrases for hand-editing the files, remembering to comment them out before attempting deployment of the web application.



## Ant Build Process

Though most students will be happy to let the **ant** command take care of things, some students (and most instructors) will want to understand the inner workings here a little better. Each web project in the course has its own **build.xml** and **build.properties** files; these rely on the master environment variable **CC\_MODULE** to import targets and properties in a central directory, **%CC\_MODULE%\Ant**. Information here defines a routine for building a Spring application; we won't document this in detail here, but rather summarize the interesting tasks:

- Clean out and create a **build** directory, which becomes the root of a web application (cleanup is a little tricky since Tomcat likes to lock certain files)
- Compile Java source files to **build\WEB-INF\classes**
- Copy resources from **docroot** to **build** – this may include HTML and JSP, the **web.xml** and Spring configuration files, message resources, and custom tags
- Copy the Spring JAR into **build\WEB-INF\lib**
- Create a Tomcat context file, e.g. **LandUse.xml** for the LandUse application, and copy it into Tomcat's **conf\Catalina\localhost** directory

Not everyone is familiar with this last step, or with this approach to deploying applications to Tomcat. Note that there is no WAR file, nor do we truly deploy our code to Tomcat. Rather, we use the context file to tell Tomcat where to find our application, and to map it to a certain context-root URL. This has the advantage of keeping our build entirely local and letting Tomcat reload as necessary when we change a JSP or class file. It can lead to some confusion, though: once the context XML is in Tomcat's tree, a build failure will leave Tomcat thinking that the application is deployed when it's not even well-formed. This may manifest as exceptions showing in Tomcat's console on a later startup. It's typically harmless, and easy to fix: just rebuild the application completely, from any of its example, demo, or lab steps; the build will overwrite the current context XML to point to itself, and Tomcat will start serving files from that recent build.



## Eclipse Overlays

Capstone Courseware provides an optional package of workspace and project files for Eclipse WTP 1.5 for this course. (See the course Setup Guide for download URLs.)

Instructors, use this package on your own initiative and at your own risk. You should have significant experience yourself with Eclipse WTP before using the overlay package in the classroom. The workspace and projects have been tested lightly with the course but are not part of the standard product.

That said, this overlay should save a good deal of work for those who wish to use Eclipse instead of the text editor and command-line tools that are standard for the course. See the documentation inside the workspace itself for general notes on usage: when you first open the workspace you'll see a **ReadMe.txt** that directs you to deeper, HTML-based documentation. Be prepared to walk students through the first few exercises in Eclipse; the notes in this file are for experienced Eclipse users, and will not be clear to many students on their own.

See also the file **Spring Module Notes.html** (available in the IDE as well) for specific matters related to using Eclipse WTP with this course.

Again, Capstone Courseware can only offer complete technical support on the standard course, and while we hope this overlay is convenient, it is not as thoroughly tested as the core lab image at this time. If a given exercise is giving trouble, please be certain to build and run it from the command line, using the SDK tools as prescribed in the student guide, before contacting Capstone.





## Teaching Notes

### Chapter 1

This chapter should accomplish two things: give students a straightforward overview of Spring, and prove the machine setups by building one each of the two primary types of applications in the course, which are a standalone application that uses a bean factory and a web application that uses a web context configuration. Along the way students can get familiar with the layout of course exercises. Don't let inquisitive students bog you and the rest of the class down during these two examples; there's plenty of time to delve into all the details in the next few chapters, and you'll be seeing both of these applications again.

### Chapter 2

Here we start building the foundation of Spring IoC, which will gradually become central to course exercises over the remaining chapters, as it is central to most Spring development. At first the beauty of the bean factory, inversion of control, and dependency injection may not be obvious to everyone. By the end of the class it probably will, but in the meantime, try to promote the idea that while J2EE gives a set of IoC features to a select few types (servlets, EJBs, etc.), Spring is egalitarian: it gives these same features to any Java class. How nice to be able to configure any class you write with properties that might not be known at development time! It's the end of the properties file, ad-hoc system properties, and so forth, and much more elegant and powerful than any of those hoary mechanisms.

An interesting side note that you might make during or after the Singleton vs. Singleton exercise: in order to enforce singularity (or prototype behavior) even over the rules set by a Java developer, Spring has to carry out some privileged actions. For this to work, either there must be no security manager in place, or it must have a policy that allows the Spring codebase some special permissions. A rough list is here:

```
permission java.io.FilePermission "", "read";
permission java.io.FilePermission "-", "read,write";
permission java.util.PropertyPermission "*", "read";
permission java.net.SocketPermission
    "www.springframework.org:80", "connect";
permission java.lang.RuntimePermission "accessDeclaredMembers";
permission java.lang.reflect.ReflectPermission "suppressAccessChecks";
```



## Chapter 3

Though technically it introduces the distinct idea of dependency injection, this chapter is largely an extension of Chapter 2. We carry our study of Spring IoC to completion, and by the end we should be able to construct arbitrarily complex graphs of objects: for the business tier, the web tier, or the persistence tier, later in the course. Remember the Spring IoC mantra of using the lightweight container to “create, configure, and assemble.” Chapter 2 is about “create,” and a little about “configure;” this one’s more on “configure” and a lot about “assemble.”

One note on the Policy demo: to accommodate the different concrete paths at which the code is deployed, later examples will rely on the `CC_MODULE` environment variable, and various beans will on their own initiative expand the phrase “`${env.CC_MODULE}`” found in their property values by looking up this variable. But that seems a bit much for this exercise, and so the exact path of the code source must be entered when configuring the policy. This is simple enough and the path is shown in the demo instructions. But the demo answer will have a path from the **Examples** tree, so any copy-and-paste solution will need some additional editing. It also means that course deployments to non-standard roots (other than `c:\Capstone`) will require editing of the XML file, even for the answer code.

## Chapter 4

This “mini-chapter” on validation is the first of three new chapters created in an effort to modularize the material and make learning more incremental. Validation is indeed a core-module feature in Spring – but it is also clearly distinct from the IoC/DI/factory material in the previous two chapters. This chapter sets up nicely for the later Chapter 8 on binding and validation in web applications, but it should be of interest outside the web tier as well.





## Chapter 5

This chapter attempts to introduce the Spring web module as simply as possible – which is not very simply, but still it should be manageable. Successive chapters flesh out the web module, but in this one we establish a “kernel” of the dispatcher servlet, handler mappings and controllers, view resolvers and views. The initial exercise refactors a servlet application – which means we don’t see much in the way of new features from the application during the demo, but it also helps us start on familiar ground and see what Spring MVC is at its most basic. The example at the end of the chapter should serve as a good summary of the basic concepts; this seemed necessary since we’ve thrown so many new and important pieces at the student in quick succession. The lab might be a bit grueling but it’s important to do that first Spring web app more or less from scratch, and by the end students should have a pretty good handle on the basic concepts and construction techniques.

In this chapter we start to use a home-grown notation for page flow, as a way of illustrating various application designs succinctly. In Course 117 we spent a few coursebook pages explaining this notation, but (a) it seems pretty close to self-explanatory and (b) the long explanation seemed to take the wind out of the sails in the middle of a chapter. The basics are that page symbols are views and rectangles are controllers; UML class symbols are used for JavaBeans and command objects; solid arrows are request paths and dashed arrows are forwards to new views. Labels on arrows represent different buttons and/or requests and different control outcomes.



## Chapter 6

In developing this course from Course 117, this “mini-chapter” was separated from the previous one to isolate what are really not fundamental techniques but options and variations. On the other hand, it is sensible for a Spring developer to at least be aware of these choices – different HandlerMappings, ViewResolvers, etc. – before getting too deep into a development effort. This is why this chapter appears here and not closer to the end of the course. Still, it is probably the best candidate for cutting, if students might benefit from the additional time spent on other chapters. Most of this material could be derived by API study after class – though the redirect techniques are probably worth a quick look in any event.

You may well have your own (and perhaps contradictory) advice to offer at the end of the chapter, in the “Which Way to Go?” discussion; so much the better. This is entirely subjective stuff, and if there were one best way to map URLs to controllers or to manage the dependencies between controllers and views, then presumably Spring would stick with that and leave out the others.

## Chapter 7

This chapter and Chapter 9 are tightly linked, and it was tricky to figure out exactly how to divide and organize these topics. Basically we’re trying to establish command objects first, and the full flow orchestration that comes with form controllers later on. Then, the **MultiActionController** needs a home, and this seems like the right place, especially as it gives some exercise in dealing with command objects and validators more or less manually.

**MultiActionController** is a bit of an odd duck, actually, living off to the side of the main line of inheritance in the servlet-MVC package. This and Chapter 9 are mostly a long walk down that inheritance tree, and that makes this class a sort of side road.

It may surprise students that, after all we’ve said about dependency injection, command objects are not injected. Instead, only their class name is given. Any questions on this present a great opportunity to talk about bean scope and how a singleton controller couldn’t very well support an injected command-object dependency. (Still, some may argue, wouldn’t an **idref** dependency make sense here, with a prototype **scope**? Spring MVC probably could have gone this way ... but for better or worse it didn’t. Request **scope** is another interesting road not taken – and in my view it’s an indictment of Spring’s web-scoped bean feature that they didn’t take advantage of it in their own controller classes.)



## Chapter 8

This is the third of the “mini-chapters” chipped off of bigger chapters from Course 117. Here we’re trying to focus on binding and validation in isolation from form controllers. We can’t really segregate these topics completely but at least conceptually this chapter should be self-sufficient. The only real concession is that, for binding to work in both directions as one would expect, a form controller needs to be in play. We leave this as a clear hole in the story, forward-referencing the next chapter and even leaving this chapter’s lab exercise in a less-than-complete state pending the introduction of Spring custom tags.

It’s probably fair to say that the failure to handle lowercase values in the LandUse demo on editing enumerated types is a bit of a straw man. HTML `<select>`s and `<option>`s allow for labels vs. visible text and this is probably how the problem here should be handled. However, the ensuing lab exercise makes a more compelling case for taking control of enumerated-type binding, and so it seems like a good flow, though the sharper students may challenge that first demonstration.

One interesting side note for this chapter concerns JSF, which is of course a big competitor for Spring MVC. In JSF, “converters” not editors are used, and on one hand they seem to throw away JavaBeans’ **PropertyEditor** standards. On the other hand, one can perhaps see why: JSF converters don’t rely on a base class like **PropertyEditorSupport** and sport just two simple conversion methods, one for each direction. Arguably, Spring’s property-editor registrars and more general IoC features make configuration of property editors more manageable than JSF’s converters, which are easy to configure as part of a view but then this system doesn’t scale up as well for large applications.

Spring vs. JSF validators is more a question of scope, since Spring validators target objects and JSF ones target individual values. Each approach has its advantages; I happen to prefer the object scope, and for practical reasons but it’s very nearly a matter of taste. The arrow does point to Spring regarding message localization and management. Finally, any discussion of web validation should probably include a mention of the Struts Validator – neither Spring nor JSF has anything quite as ambitious, and integrating it may be a matter of interest to advanced students.





## Chapter 9

In this chapter, we try to tackle just one main job, which is managing HTML forms with form controllers – but it's a tricky task, to say the least. The understatement of the whole coursebook may be that **SimpleFormController** isn't really simple. On the one hand this class is the crown jewel of the MVC library, and has great facility once you get control of it; on the other hand it's a remarkably hard beast to tame. The demos are intended to walk students through the many necessary steps so as to make the task not too daunting. Otherwise, this is one of those tools with which it's best to work from a successful example and start varying things, rather than build from scratch and watch things fail and fail until suddenly they work. Be sure you know these demos very well before tackling them in class.

By the way, this may be the right place to point out a custom tag that's been in use in the course examples since Chapter 5. There's enough going on in that chapter that it seems best to leave `<cc:springBean>` undiscovered for a little while; but if students ask, or once you reach this point in the course, you might want to dig into it and explain what it's doing and how it's doing it. The idea here is to bring Spring beans into a JSP – even if that JSP doesn't enjoy the services of the Spring MVC framework. So for the database wrapper in LandUse, and similarly in the Wholesale application, the JSPs use this tag to get access to an object that all the other components in the application load using the application context.

## Chapter 10

This chapter closes out coverage of the web module, and it may be seen as a bit of a hodgepodge of advanced topics. But there is a clear theme – or, at least, there are a few! Exception handlers and interceptors more or less of a piece, and the context-and-lifecycle section seems to put the finishing touches on the idea that any JavaBean can have the first-class treatment of a contained object. But ultimately this is the stepping-off point, and towards the end of the chapter we're mostly telling students where they might look next as they continue to learn about Spring.





## Chapter 11

The highest value pieces of this chapter are the JDBC template and the declarative transactions support. We do a lab in one and finish with a demo in the other; we could get deeper into transactions, but more and more it seems that any lab that concludes a course is doomed never to be done by students; so we're trying here for a mid-chapter lab that will get done, and a demo that makes a nice closing act for the instructor.

Though Hibernate is an obvious and popular choice for a persistence framework to work in tandem with Spring, we give it relatively light coverage here. There are two reasons. One is that we have a separate course in Hibernate and figure that students will either be on their way to that course or will know Hibernate already. The other is that the course tool set is already laden with Spring, MySQL, JSTL, and more; adding Hibernate would have packed another 10 megabytes onto the lab installer, and the benefits would have only applied to at most one lab exercise and only been available to those with some previous Hibernate experience. So we keep it simple, but instructors savvy with Hibernate may well want to expand on this chapter with some alternate DAOs and other exercises.





## Revision History

**Version 2.0.1** is the initial release of the course, and is created with Course 117, version 2.0.1, as a basis. In responding to instructor feedback on that course, we've added several examples, demos, and labs, and have attempted to ease the way for students who found too many steep ascents in 117. Major changes are as follows:

- The DOS scripts and Ant build files now rely on the variable **CC\_MODULE**, not **CC\_HOME**. This is in keeping with other recent Capstone courses and has simply proven to be a more potent and maintainable approach. You shouldn't see any complications in this switch, as in both old and new versions use of the variables is baked into the existing files.
- Several example trees have been renamed and/or expanded. **Ellipsoid** is now **Ellipsoid/Web** so that we can explore basic IoC techniques in **Ellipsoid/Beans**. **Beans** was not real descriptive and is now **Bank/Beans**, and there is a small **Bank/Web** example as well. **Factories** is now **Parsers**, to be more in line nominally with **Transformers**. **Collections** now has two steps, with **Step1** being the old example and **Step2** bringing in some interesting techniques using the utility schema.
- The **Bank/Beans** example now has two additional steps, and these form the first exercise in the use of factories, with a new **BankManager** system in place.
- The old demo on factories was a bit much for some students, since it used JAXP parsers and not everyone knows those; this has been refactored to an optional lab.
- The **Transformers** example has been explained in some more detail, and extended considerably to create a major new lab in Chapter 3.
- The old lab on validators is now the second of two, with a new lab on building the basic **OrderValidator** as the primary exercise. Students mostly found the **ListOfBatchesValidator** to be too much all at once; now it's additional exercise in basic validation and also illustrates nested paths – which have gotten more treatment in the lecture material as well.
- The initial web demo in refactoring the **Ellipsoid** application has been enhanced a bit to make it more obvious how things are changing as Spring features are introduced.
- There is a new, very simple web application in **Bank/Web** that should help to cement the basic concepts of Spring MVC. This is an example at the end of Chapter 5 and provides work in property editors later as well.



- We've moved away from Apache Derby and now use MySQL 5.0 as the RDBMS for the last chapter on persistence. This brings the course in line with other Capstone materials, notably our JDBC and Hibernate courses. It also removes any need to restart Tomcat during that chapter's exercises; Derby could only support one connection in embedded mode as we were using it, and this made some of the later demos rather clunky.





## Troubleshooting

If you run into any trouble with code exercises, the first and best thing to do is to double-check that the classroom machines have been set up precisely according to the course setup guide. Especially, the wrong version of a tool can cause significant problems; don't wander off-book in this way unless absolutely sure you can support the software that you prefer and that we haven't tested. Check environment variable settings carefully, too; these are the cause of a great many classroom glitches.

Below are some specific pitfalls that have come up in previous offerings of the course:

- If you set the **CC\_MODULE** environment variable in an individual command console (and not globally, e.g. through the Windows Control Panel), you must be sure to start Tomcat from that console or from one that also has the environment variable defined correctly. This is necessary because some Java classes running in the deployed web applications will attempt to resolve the value of that variable, as a way of finding resources outside of their own web contexts. A good strategy is to set your environment in the console where you'll do your Ant builds; **start** another console, which inherits that environment, and change to the Tomcat **bin** directory in that second console. Use the second console to **startup** and **shutdown** Tomcat as necessary.
- Attempts to validate Spring beans XML files against the schema will succeed only if there's an Internet connection, since the **schemaLocation** attribute points to the public schema. If the classroom is not networked, you can reset this attribute to the schema cached locally in **%CC\_MODULE%\..\Tools\Spring2.0\dist\resources**. (This change would have to be made for each XML file, which is a bit clunky, but once you've done it, it's easy to do again by copy-and-paste.)



## Errata

At this time there are no errata for this version of the course.





## Feedback

We very much appreciate whatever feedback we can get on our courseware – especially from the instructor's perspective. Naturally, the more specific, the better, and we strongly encourage you to make notes on issues you may encounter in the classroom, whether they're typos, missing files, or suggestions for clearer language to explain a concept. We can't guarantee that we'll act on every suggestion, but we're aggressive about stamping out problems and try to be highly responsive. Hopefully this means that when you give us good feedback, you get a better course the next time you need to teach it.

Please direct all courseware feedback to

Will Provost  
Capstone Courseware  
<mailto:provost@capcourse.com>  
877-227-2477

For anyone who's interested, we have a very informal defect-tracking system, based in Excel spreadsheets with columns to capture defect location, nature, status, and author feedback. Ultimately, feedback goes into these sheets, so if you want a template, we'll be happy to provide one, to facilitate the reporting process.

