



**Capstone Courseware, LLC**

33 Boylston Street  
Jamaica Plain, MA 02130

877-227-2477  
capstonecourseware.com

# Spring Web Flow

**Version 2.0**

## **Instructor's Guide**



## Overview

This course combines three main parts: an initial overview of Spring as a whole; a mini-unit on Spring Web/MVC; and then the bulk of the course consists of the final unit on Spring Web Flow itself. So, while Web Flow is the main target of the course, we spend the first couple of days on Spring proper, since the one is built on the other and effective Web Flow development will naturally involve middle- and persistence-tier objects configured in a Spring web application context.

The first module develops fluency with basic instantiation, configuration, and assembly of objects of various stripes using bean factories and application contexts. It also introduces other concepts that are useful to the Web Flow developer, such as lifecycle hooks, annotation post-processing, component scanning, and the validation framework. Lecture material considers practical applications in the web context, but exercises in this module are simple, standalone Java SE applications.

The mini-module that follows gives students a basic working knowledge of the Spring Web module: **DispatcherServlet**, **HandlerMapping**, **ViewResolver**, and so forth. The most important thing for purposes of this class is to understand how Spring web applications use Spring beans files to configure their web application contexts; this then is the gateway to the Web Flow registry, executor, handler mapping, and handler adapter, and Web Flow expressions may well refer back to other beans found in the application context.

With this foundation understood, we move into Web Flow itself. Web Flow of course offers a radically different approach to building web applications than Spring MVC – or Struts, or JSF, or really anything with which the student is likely to be familiar. (Of the frameworks just mentioned, Web Flow probably most closely resembles JSF.) So we take an incremental approach, working hard to control the pace at which new concepts are flying at the student. This means giving a stiff arm initially to even such basics as actions and EL; but experience has shown that jumping right into these along with view states, transitions, and broader understanding of Web Flow coordinates request processing, is just too much all at once.

By about the middle of the fourth chapter, the student will have seen just about all of the key concepts and techniques required to make real headway with building a Web Flow application. We then start into intermediate and advanced topics, starting with the latter part of Chapter 4 in which we confront the weird relationship between Web Flow scopes and Spring singleton scope, and moving into converters and validators, subflows, and lifecycle listeners.



## Timeline

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

### Day 1

2 hours	Module 1, Chapter 1
2½ hours	Module 1, Chapter 2
2 hours	Module 1, Chapter 3

### Day 2

2 hours	Module 1, Chapter 4
2½ hours	Module 1, Chapter 5
2 hours	Module 1, Chapter 6

### Day 3

2 hours	Module 2, Chapter 1
1 hour	Module 2, Chapter 2
2 hours	Module 3, Chapter 1
1½ hours, spans days	Module 3, Chapter 2

### Day 4

2 hours	Module 3, Chapter 3
2½ hours	Module 3, Chapter 4
2 hours, spans days	Module 3, Chapter 5

### Day 5

1½ hours	Module 3, Chapter 6
2½ hours	Module 3, Chapter 7
1½ hours	Module 3, Chapter 8

If time is tight, the best candidates for cutting are the final chapters of each of the first two modules. Better to drop these and make more time for the Web Flow module as a whole.





## 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.

Projects in the first module contain only standalone applications. In these, the Ant build will:

- Clean out and create a **build** directory
- Compile Java source files to **build\classes**

Then, the provided **run.bat** in each case will launch Ant again, but this time with a specific target that launches the application. Properties **app-class** and **args** inform this launch, though for many of the projects the settings are the same: the application class is simply "Controller" and there are no arguments. Some applications do accept command-line arguments, and in this case the **run.bat** will pass those from the actual command line along to the launched class. The necessary Spring module JARs are included in the class path in all cases.

Starting in the second module, and in the third module as well, Ant will build and deploy web applications to a Tomcat server. For these projects Ant will:

- 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\classes**
- Create a staging area for assembly of a web application
- Copy resources from **docroot** to that area – this may include HTML and JSP, the **web.xml** and Spring configuration files, message resources, and custom tags – and copy compiled Java classes to its **WEB-INF/classes** folder
- Copy the appropriate Spring module JARs into **WEB-INF/lib**
- Build a WAR file from the assembled tree





## Eclipse Overlays

Capstone Courseware provides an optional package of workspace and project files for Eclipse Ganymede for this course – also using WTP 3 and Spring IDE 2.2. (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, and Spring IDE 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 module notes file in each workspace – e.g. **Spring Module Notes.html** – 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 – Module 1

### 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. Try not to let inquisitive students bog you and the rest of the class down during the example; they'll see it again almost immediately in Chapter 2 and there will be plenty of opportunity to dive into the details.

### 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 Java EE 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";
```

The lab in this chapter is not all that challenging, but it's the first one and should get everyone rolling without much trouble and build familiarity with the <bean> construct.



## Chapter 3

The previous chapter focuses on instantiation, but does bow to the necessity to allow some basic configuration via `<property>`s in order to make the exercises at all interesting. In this chapter, we expand on both ideas, with options to configure via constructor arguments and also new ways to customize object instantiation using existing factory implementations. This starts to tumble out as a promotion of all the different ways in which we can build objects using Spring. It may seem a bit scattered after a while, or like overkill, but it's good to note that Spring is really just trying to be as adaptable as possible to various common object-model designs: existing factory methods and abstract factories, immutable objects that can only be configured via constructors, and so on.

## Chapter 4

Now we start to perform dependency injection for real. With single-value configuration as we've seen it in earlier chapters already understood, this feature should come very easily. By the end of this chapter we should be able to construct arbitrarily complex graphs of objects.

This chapter also introduces two features that may land with a bit of a thud in some classes: autowiring, and auto-detection of annotated component classes. The previous chapter did introduce component scanning, so that we could consider `@PostConstruct` and `@PreDestroy` hook methods. But now we look at both of these features in detail.

Autowiring is controversial: used heavily by many in the Spring community but reviled and labeled outright bad practice at many companies. We try not to advocate one way or the other here, but simply to cover the feature completely and accurately.

Component scanning and annotation processing is a different choice entirely, although this and autowiring are often practiced in concert. Like autowiring, component scanning seems to tear away some of the declarative benefit of Spring; annotation processing is another matter but where the kingpin annotation seems to be `@Autowired` we do wind up in the same boat. It might be worthwhile to point out that other sorts of annotation post-processors can enable features that even the anti-autowire crowd would find more appealing, such as `@RequestMapping` for the web tier or `@Transactional` for DAOs.





## Chapter 5

Here we expand on our configuration and assembly capabilities, with an emphasis on configuring collections and maps but also some further consideration of reuse mechanisms including **util:property-path** and **util:list/set/map**.

One note on the Policy demo: to accommodate the different concrete paths at which the code is deployed, code in other courseware modules often 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. Finally, note that this syntax is demanding: the trailing slash matters, and even incorrect case for the drive spec (“C” vs. “c”) will result in a failure to match the permission. Read and type carefully!

## Chapter 6

Validation is indeed a core-module feature in Spring – but it is also clearly distinct from the IoC/DI/factory material in the previous four chapters. In the materials we applaud Spring’s recognition of validation as a core feature: not something tied to the web tier, for instance, though Spring MVC does a nice job of extending the core validation mechanism for HTTP request parameters, HTML forms, and web controllers.





## Teaching Notes – Module 2

### Chapter 1

This chapter attempts to introduce the Spring web module as simply as possible – which is not very simply, but still it should be manageable. 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. For purposes of this course, this is usually sufficient. We've included a lab at the end of the chapter, for groups that have more interest in Spring for web applications, but it's flagged as optional, and in the typical timeline it will be skipped.

### Chapter 2

We've separated this chapter 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 interesting for general purposes, but we include it here to reinforce understanding of the relationship between Web Flow and the Spring Web module. Of course, the Web Flow handler mapping will be a new choice, seen in just another chapter of the course, and when the flow executor selects a view it will hand control back to the **DispatcherServlet** to resolve and/or render that view.





## Teaching Notes – Module 3

### Chapter 1

Go slow! Even for experienced web developers, it takes a while to adjust to Web Flow's vision of a web application. We introduce the project here and start by putting it in context assuming that it will be stacked on top of Spring and the Web module – while pointing out that Web Flow can work over JSF instead. We try to begin at the beginning, and explain how Web Flow as a whole is plugged into the Spring Web module and the **DispatcherServlet**, and to show how the flow executor comes to handle a given HTTP request. Then, within that bubble, we give a basic idea of how flows are organized, why they are what they are, and of the most basic building blocks, which are states and transitions.

This is usually plenty for students to chew on for a little while, and we move quickly to a simple example, which is a familiar application re-implemented in Web Flow. This at least should clarify the mechanics of flow registry and flow executor, and give students a taste of flow definitions. We wrap up with an attempt to summarize the value proposition, as for many students it won't yet be apparent why we would go to these lengths to build a web application when Struts or Spring or JSF or just servlets/JSP models are available. By the end of the course this case should be made pretty well, but it'll be good to make a promise or two at this point as to the value of Web Flow, to help motivate the work that lies between here and there.

### Chapter 2

Now we drop down to ground level and start working through different parts of the flow definition language: how to configure views and transitions, and how to use them effectively. For a little while we're going to be building some artificially simple applications, such as the Wizard exercise at the end of the chapter, because we're trying to keep the presentation of the Web Flow model as un-cluttered as possible and to focus on just a few bits at a time. So, no actions, no EL, not really even any model beans for now. The hardest part of teaching this chapter may be explaining this to the students, that there's more to the typical Web Flow application but that we're working bit by bit for now. The exercises themselves should be straightforward.



## Chapter 3

Now, having worked with states and transitions in isolation, we introduce variables, Web Flow scopes, actions, and use of EL. This is where students start getting the chance to build some more practical solutions with Web Flow: the Wholesale **define** flow as a demonstration, and the LandUse summary-to-detail transitions as a lab exercise.

I like to set up the demo as a lesson on how much EL is too much. The use of EL is a major differentiator for Web Flow, as it enables the use of controllers, services, and other Java objects without any further need to refer to things like HTTP requests and responses, errors objects, etc. And flow decisions can be encoded cleanly in the XML definition, rather than having to live partly in various controllers and partly in view-resolution logic. But there is the temptation to overuse EL, and the Wholesale application has indeed taken a step too far in this direction. The failure to call the right overloaded Java method is just one outcome of a larger mistake, which is to try to use EL in flow definitions to implement true control logic. EL is great for flow logic, and for invoking control logic; but it should end there.

Also, at this point, the exact order of events inside the flow executor starts to be interesting. We will not try to develop a detailed understanding of how the executor does its job – that would take a lot more time than we have and would require investigation of the source code. But a simple tracer such as the example at the end of this chapter should shed some light on the inner workings, and in ways that are relevant to application development: e.g. what happens first/second/third in a transition that has an action on a state that also has an exit hook? When are flow-scope beans initialized? Etc.





## Chapter 4

The title of this chapter suggests that we've not really considered Web Flow scopes already, when in fact we have. This chapter does flesh out understanding of flow, view, and flash scopes; but the point is more to understand some idiosyncrasies of Web Flow that have to do with scope management, serialization, and its relationship with Spring and HTTP URL mapping.

The rhythm is a little odd in that we jump right into an early lab, and then see another one about mid-way through, and finish the chapter with examples and demos. Part of this is looking to give the student more exercise with basic Web Flow authoring, so as to get more confident with these core techniques. But each of the labs also addresses some new feature or area of concern: dealing with multiple forms, request parameters and flash scope thanks to POST-REDIRECT-GET. The scopes demo in-between should provide a nice clear summary of scope options and behaviors.

Most of the latter part of the chapter (after the second lab) deals with a truly surprising set of behaviors in Web Flow when using Spring singletons. There are a few ways to involve Spring beans in Web Flow applications; but, as the main demo in this part of the chapter shows, there are really only two safe practices: reference to beans at the beginnings of EL expressions, and **@Autowired** dependencies from beans that live at Web Flow scopes such as flow or view scope. This is a bit of a brain-bender, and it's a good idea to be very familiar with this one before launching into it, and to take it slow with students as well.





## Chapter 5

We take a step back in this chapter and try to consider the process of data binding more comprehensively. First, what sorts of HTML form controls can be bound to what types of properties on model beans? The initial example gives a good sense of that, including some issues with checkboxes and with date and time values.

Then we look at custom converters and the idea of a “conversion service” that loads and uses these converters. This has actually been in play for the LandUse application all along, and we did mention it briefly before but only from the perspective of Spring beans configuration. Now we delve into the **Converter** and **ConversionService** API, and see the typical technique for putting one’s own conversion logic into play.

The last section about dependency-injecting converters as Spring beans is not absolutely necessary, but it does offer a good perspective on best practice and a deeper look at the **ConversionService** interface while we’re at it. The resulting “injectable” conversion service is used a lot in the remaining exercises, so even if you don’t go into it in great detail it’s worth a quick look.

## Chapter 6

Here we tackle the problem of input validation, and see Web Flow’s somewhat bizarre but ultimately simple mechanism for plugging in validation logic. It doesn’t look like much at first, and the heavy reliance on naming conventions could use some re-thinking. Rather than “convention over configuration,” Web Flow here is practicing “convention ... period.” Later in the chapter we do get a nice surprise, seeing that it’s not actually necessary to write out distinct **validateXXX** methods for the same model bean as used by different view states; and that takes away the one issue with the convention-based mechanism that would really have an impact on code efficiency and reusability. The rest is more a matter of style, and while the Web Flow approach takes some getting used to (and doesn’t integrate all that well with other sorts of validation, even Spring **Validators**, which is disheartening), it does work just fine.



## Chapter 7

Up to this point we've been staying carefully inside the Web Flow bubble: all the interesting application logic has been contained within a flow, rather than having anything to do with the start or end of that flow. In fact most of our applications have been single web flows, entirely self-contained – but with the odd behavior in some cases of restarting after the final page is submitted, since that's what a flow will do by default when it reaches an **<end-state>**.

Now we “zoom out” a bit and look at the flow as just one piece of a bigger puzzle. First we consider the input and output contracts that can be defined by a flow, and how these can be used by non-Web Flow applications. We discuss **FlowHandlers** but don't get into any hands-on exercises in this.

More interesting for our purposes is the flow as part of another flow – that is, the master flow / subflow relationship. So here's where the chapter title finally shows up! and we take a few passes at understanding what subflows are and how they can be harnessed. It turns out that it takes a fairly complex case study to make subflows really appealing, and the first few examples are all compromised in some way. The Wizard has to fudge its URLs and image names on the second pass through the subflow; LandUse is a little cleaner but there's not much we do with the subflow that we couldn't have done with a simple view state; and then the tracing application has trivial application logic.

Fear not, for the rest of the chapter is occupied by the Travel application, which should be plenty big and complex to satisfy students' appetites. We see one of its flows acting on its own early in the chapter; at this point we're able to consider the whole kit and caboodle, and finally to see some of the practical advantages of subflows and input/output contracts.

This chapter's lab exercise is a sort of grand finale (undercut perhaps by the niggling detail that it is not, in fact, the last lab in the course!), in which the student will need to carry out a significant amount of flow configuration work to build that **main** flow from scratch, while bringing the **<subflow-state>** front and center for this exercise in particular. As mentioned in the **Timeline** section, it's worth it to leave plenty of time for this exercise so that students can really stretch out on it. Having built this one, they should be feeling pretty confident that they can go and do real work with Web Flow after the class is over.





## Chapter 8

At the risk of anticlimax, we include this last chapter to point out one remaining feature, which is the flow execution listener. In one way this is the tip of an iceberg: there's a lot more to the Web Flow execution API once we've peeled back the covers. But this particular interface can be a real help at the application level, and is worth considering along with just a few of the types on which it depends (request context, definition objects), even if we leave the API documentation alone for months after class is over. The final lab should give a pretty good idea of what sorts of generic features might be implemented for an application by observing the behavior of the flow executor.

## Revision History

**Version 2.0** is the initial release of the course. It includes materials from the 2.5 releases of Courses 117 and 117A, which have longer histories going back to Spring 2.0.





## 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\Spring2.5\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 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.

