
Spring-Web
Lab Instructions
spring
source

This Page Intentionally Left Blank

Table of Contents

Chapter 1. rewards-online: Getting Started with the Development Environment

1.1. Introduction

This lab will introduce you to the suite of reference applications you will work on throughout this training course. Your goal is to learn the end user requirements of these applications and the overall system architecture. After you have a solid understanding of the architecture and functionality that must be implemented, you will be in a great position to dive into design and implementation details in subsequent labs.

This lab will also introduce you to the SpringSource Tool Suite, as well as SpringSource tc Server, the server used to deploy and run applications on in this course. By the end of this lab, you should understand what problems the reference applications solve, how they are architected, and how to deploy them to the server using the tool.

What you will learn:

1. How to deploy projects to the SpringSource tc Server inside the SpringSource Tool Suite
2. Background on the business domain used in this course
3. Tools for debugging web applications

Estimated time to complete: 30 minutes

1.2. Instructions

1.2.1. Getting started with the SpringSource Tool Suite

In this section, you will become familiar with the SpringSource Tool Suite and how the projects you will work on are structured.

1.2.1.1. Launch the Tool Suite

Start by launching the SpringSource Tool Suite application via `c:\spring-web-{version}\sts-{version}\STS.exe`. The first time STS is launched, the course workspace will be built and all lab projects imported. Building all projects will take some time. You will see ongoing

build activities in the status bar at the bottom right. Once building is complete, you can close the STS welcome screen and move on to the next step.

1.2.1.2. Review Typical Lab Project Structure

Within the Package Explorer, each lab is organized into a Working Set. In general, within a lab's working set there are two projects. The first project is where you will work; it is the starting point for a lab. The second project is the completed solution. This project allows you to compare your work with the work of SpringSource experts.

Quickly expand the Working Set for the next lab, `02-mvc-essentials-1`, and confirm its structure. The *mvc-essentials-1* project is where you would work and the *mvc-essentials-1-solution* project contains the SpringSource solution. Subsequent labs have a similar structure.

1.2.1.3. Review the Common Projects

A number of modules have been predefined for use in all labs. These make up the back end of an overall Financial Rewards application. Note that they are not deployed as a separate web module, but simply included within each lab.

Expand the `00-reward-network` working set and note the single *rewards* project therein. Note the different packages contained in `src/main/java`. The *common* package contains foundational types needed by the other modules. The *rewards* package carries out complex business transactions and provides access to the system's relational database.

Tip

You may decide to close projects you are not currently working on to make STS more responsive. If you do so, do *not* close projects in 00-reward-network or Other Projects working sets, or your code will not build!

1.2.1.4. Review the Projects for this Lab

The project for this lab, called "RewardsOnline" is organized a little differently. This is because you will not implement any code here, instead you will walk through the reference application architecture and end-user functionality.

Expand the `01-rewards-online` Working Set and note the single project. This module, *rewards-online*, is a web application that allows business to be conducted online.

As a web application, rewards-online is the highest-level module and depends on the rewards project for

several foundational types, to carry out complex business transactions, and to access data needed by several administrative screens.

1.2.2. Deploying to tc Server

In this section, you'll deploy the modules of the rewards system and bring the full-system on-line.

1.2.2.1. Verify The SpringSource tc Server Installation

A tc Server installation has already been extracted onto your filesystem within the `c:\spring-web-{version}\tc-server-{version}` directory. In this step, you'll learn how to access and manage it within STS so you can deploy projects without leaving the environment.

Navigate to the Servers view. You will see an entry for the SpringSource tc Server, which points to the tc Server installation on the file system.

Tip

If you don't see tc Server in the Servers view, try closing and reopening the view. The easiest way to reopen it is to press Ctrl+3 and type 'Servers'.

Double-click the entry to get to the server properties editor. Click on the Runtime Environment link and verify the SpringSource tc Server installation directory. When you're ready dismiss the window and close the server properties editor.

The first time you start the server you will see the following prompt:

Spring Insight weaves lightweight performance tracing into an application as it gets deployed to tc Server. Press "Yes" in response to this question and allow the server to start up. When your server is running

successfully, move on to the next step.

1.2.2.2. Deploy the rewards-online application

The RewardsOnline application is one face of the RewardNetwork, allowing rewards to be initiated using a rich browser interface. It also allows administrators to manage member and vendor accounts on-line. Since this course is about developing spring web applications, you'll be spending most of your time designing and implementing this application.

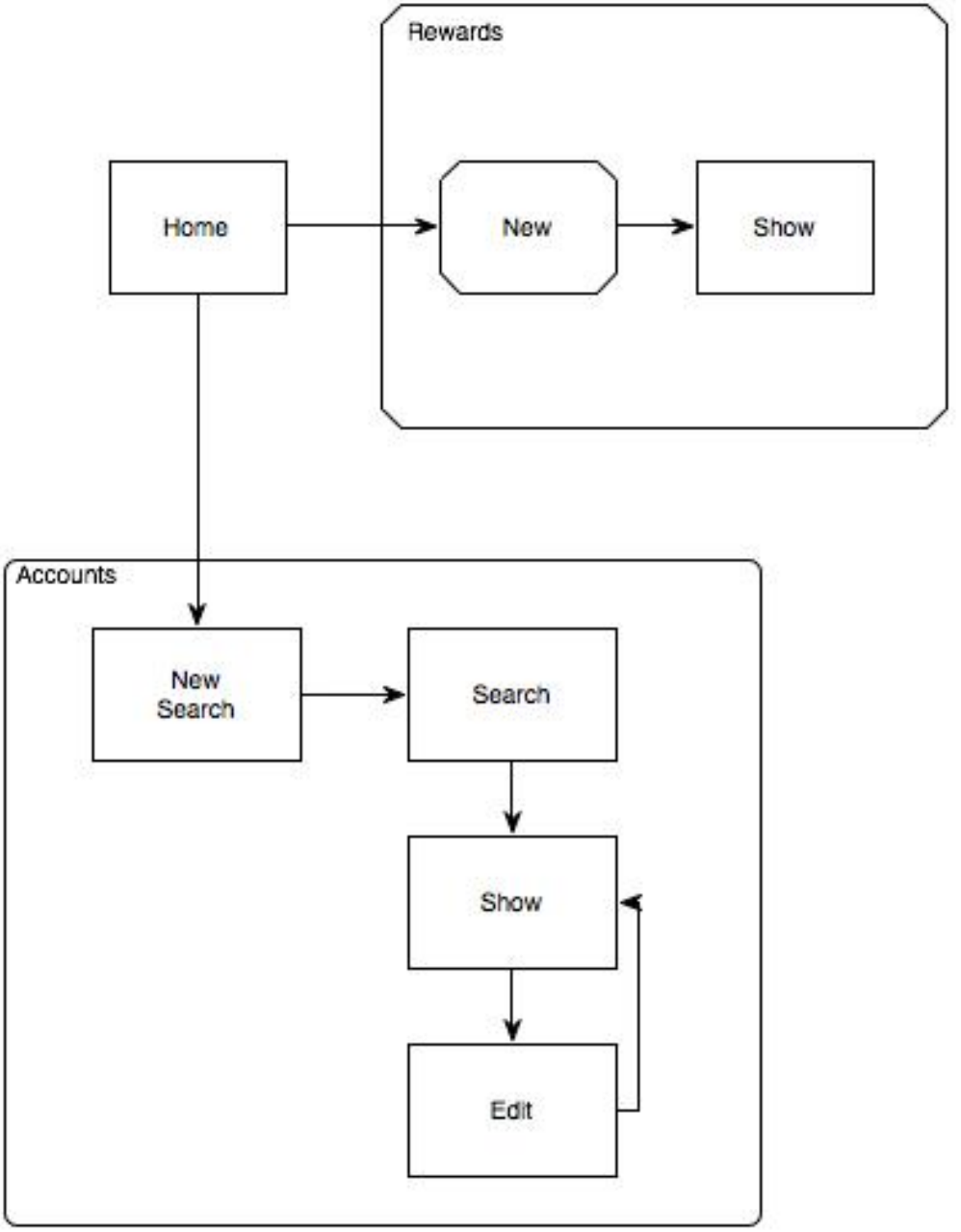
Deploy the rewards-online application in one of the following three ways: drag the project to tc Server, right-click the project and select "Run As... Run on Server", or right-click tc Server and select Add/Remove Projects. Verify it deploys successfully. Access the application at http://localhost:8080/rewards-online/login.

When you see the Login page for the rewards-online application appear, congratulations! Now move on to the next section to familiarize yourself with the functionality of the application itself.

1.2.3. Using RewardsOnline

The RewardsOnline application allows administrators to reward members for dining at in-network restaurants using a rich browser interface. It also allows member and vendor accounts to be managed on-line. Behind the scenes, this application invokes the rewards module to actually carry out transactional processing. In this section, you will walk through the on-line application. Your goal is to further cement your understanding of the reference domain, as well as the technical UI requirements addressed by this web application.

The site map for the RewardsOnline application looks like:


Rewards
Home
New
Show
Accounts
New Search
Search
Show
Edit

Notice how the web site is partitioned into two areas: in the first area you work with rewards, in the second area you work with member accounts. In the next few steps, you'll exercise the functionality across the pages in these areas.

1.2.3.1. Login

Before the application can be used, you must login. Go ahead and login from the home page by entering the mock credentials provided. When you have authenticated successfully, move onto the next step.

1.2.3.2. Create a Reward

RewardsOnline allows you to create dining rewards using a rich browser interface. Click on the "New Reward" navigation link to begin the New Reward wizard. Fill out the form provided.

Tip

Credit card number 1234123412341234 is a valid credit card in the database.

Once you have entered a valid dining transaction, select Reward. The RewardNetwork will attempt to match the credit card you entered to a member account, then calculate the amount to reward that account for dining at the selected restaurant. You will be redirected to a confirmation page that can be safely refreshed. The transaction is now complete!

1.2.3.3. Manage Accounts

RewardsOnline also allows you to manage member accounts on-line. You can search accounts, review account details, and update account details.

Select the Accounts navigation link to begin a new search. First, try simply searching for the letter "a". You should get many results back, too many for one page. Use the provided Next and Previous links to page through the results.

Now refine your search by selecting Change Search. Enter Keith as the search string. Since there is only one Keith in the database, you should be taken directly to his account details screen.

Review Keith's details and select Edit to update them. Save to commit your updates and confirm they committed successfully.

1.2.3.4. Review Internationalization

Support for multiple locales is a requirement of RewardsOnline. Test this out by selecting the Francais

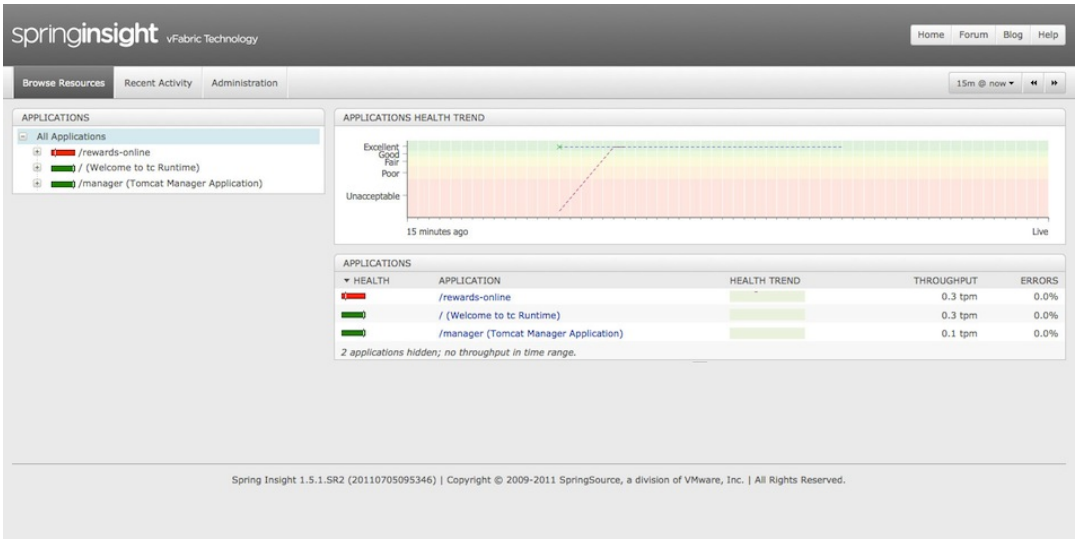
(French) link. Notice how all text in the application is now in French. The text remains in French even if you close your browser and restart it.

1.2.3.5. Review Personalization

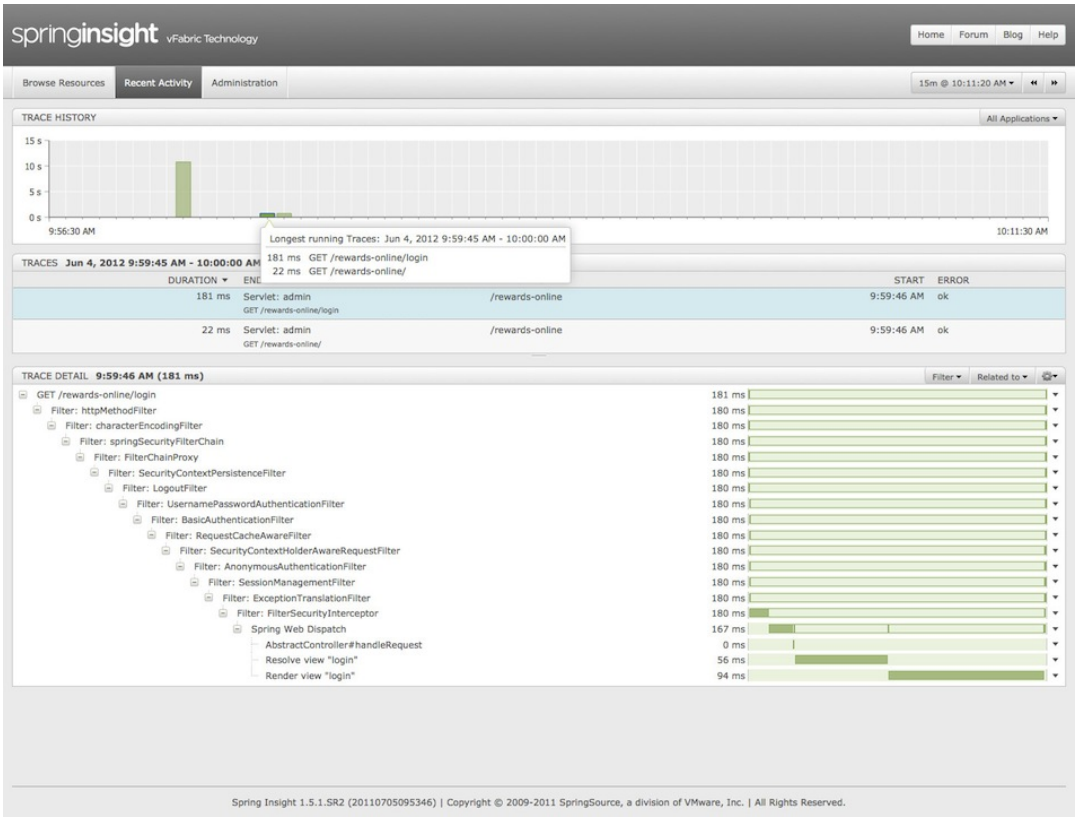
Support for personalized themes is another requirement for users of RewardsOnline. Test this out by selecting the Blue link. Notice how the skin of the application changes from green to blue. The application remains blue even if you close your browser and restart it.

1.2.3.6. Spring Insight

Right-click tc Server in the Servers view and select "Open Dashboard". Alternatively open a browser to http://localhost:8080/insight You will see the start page for Spring Insight:

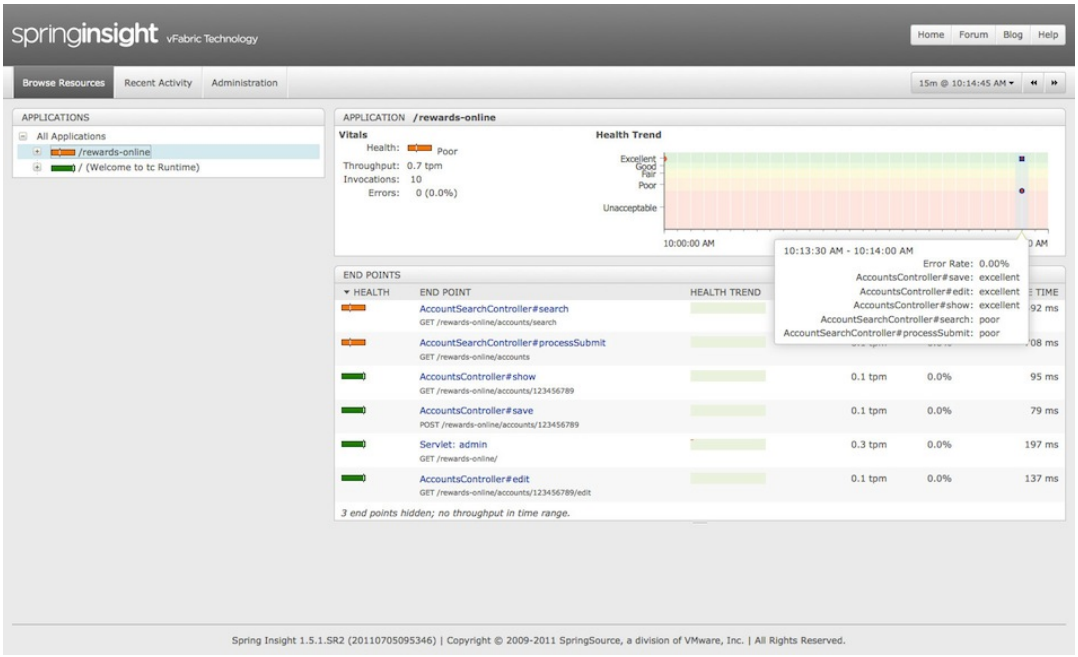


The Spring Insight dashboard presents a sequence of incoming application requests over a timeline. Under All Applications (on the left) select the name of your web application "/rewards-online". This will filter out requests for other web applications. Next, switch to the Recent Activity tab and see the same information as a bar chart. Move your mouse over any of the bars in the timeline and click it. This will show a list of traces corresponding to specific requests along with trace details as seen below:



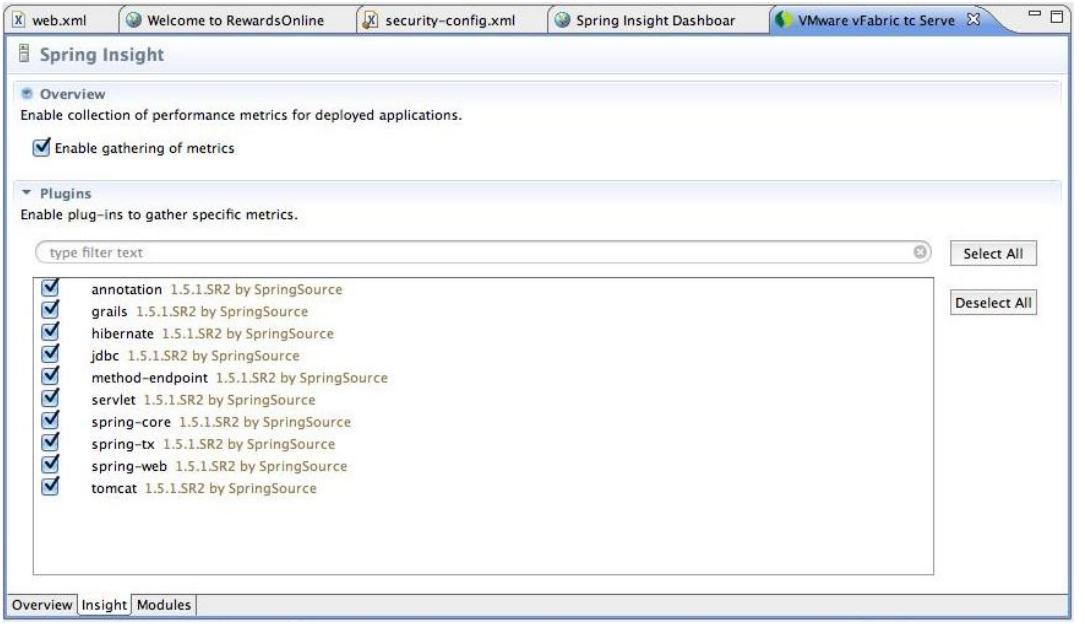
Select any request and examine the trace details. Each line in a trace detail represents an entry point into an application or a server component much like a stack trace in a debugger. Use the little arrows on the right to get more details for each line in the trace. You will get information about HTTP request/response headers, Spring MVC controller invocations, return values, transactional methods, JDBC queries and more. Within an expanded trace detail you will also see a "Go to Source" link that will take you directly to the source code.

Getting a break down for an individual request is very useful. However, at some point you will probably run a load test and will want to get the same information. Spring Insight is well suited for that because it is lightweight and doesn't alter the performance of the application. Return to the first tab - Browse Resources. On the left hand side, under All Applications you should find the current application - rewards-online. Click to show application details. You should see a screen that looks approximately like this:



This is where you can get useful information after running a load test. What you see is a list of endpoints. An endpoint is a call to a server or an application component for which you want to aggregate statistics. For example for a given Spring MVC Controller you'd like to know how often it was invoked and what was the average response time. Click on any endpoint and see a response time histogram. From here you can select a specific request that took longer than others and examine the trace details.

Before ending the lab you will see how to turn the gathering of statistics with Spring Insight on and off. Double-click the tc Server instance in the Servers View. Select the middle tab (at the bottom of the view).



In the resulting screen you will see a section titled Overview with a checkbox. Use this checkbox to enable and disable the gathering of statistics at any time during the class. We recommend that you keep it off until you need it.

Congratulations! You have completed this lab. You now have a solid understanding of the reference application architecture and functionality that must be implemented, and you are in a great position to dive into design and implementation details in subsequent labs.

Chapter 2. mvc-essentials-1: Getting Started with Spring MVC

2.1. Introduction

This lab will get you started and productive with Spring MVC. In it, you will implement several use cases of the RewardsOnline application using the MVC programming model.

What you will learn:

1. How a typical Spring web application is set up
2. How to design and implement annotated @Controllers
3. How to configure Spring to resolve JSP views
4. How to configure Spring to resolve messages from a resource bundle

Estimated time to complete: 60 minutes

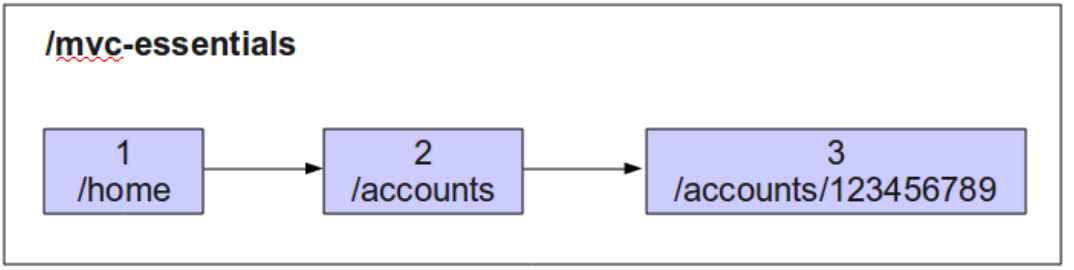
2.2. Instructions

The instructions in this lab are divided into five sections. In the first section, you will learn the functional requirements you will implement in this lab. In the next section, you will become familiar with the existing code and configuration that has been provided for you. In the remaining sections, you will actually implement the functional requirements using Spring MVC.

2.2.1. Functional Requirements

You have been tasked to implement three screens of the RewardsOnline application. The first screen should simply welcome the user and provide navigation to other top-level screens. The second screen should display a listing of all member Accounts in the database. The third screen should display the details of a single Account.

Each of these screens should be bound to a unique, independently addressable resource URL. The desired URL bindings and relationship between screens is shown on the site diagram below:



In the diagram, the outermost box represents the root of the web application, which is bound to the /mvc-essentials-1 URL. Accessing /home should display the welcome screen. Accessing /accounts and /accounts/{accountNumber} should display the (2) Account Listing and (3) Details screens, respectively. Accessing the root URL of the application should simply redirect the user to the welcome screen.

You can see this URL structure and the actual screens in action by deploying the mvcessentials-1-solution project and accessing http://localhost:8080/mvcessentials-1-solution.

When you are comfortable with the functional requirements, move on to the next section.

2.2.2. Existing Code and Configuration

Some code and configuration has already been provided for you in the mvc-essentials-1 project as a starting point. First, the overall structure of the project has been established. Second, Spring has been pre-configured to initialize your application when it deploys. Finally, the backend service you will invoke to load Account information from the database has been written for you. In this section, you'll become familiar with this existing code and configuration.

2.2.2.1. Deploy the web application

Start by deploying the mvcessentials-1 project to the server. Check the logging output and confirm it does initialize successfully.

Now try accessing the web application at http://localhost:8080/mvcessentials-1. You should receive a 404 and see the following warning appear in your logging output:

```
WARN : org.springframework.web.servlet.PageNotFound
No mapping found for HTTP request with URI [/mvcessentials-1/admin/home]
    in DispatcherServlet with name 'admin'
```

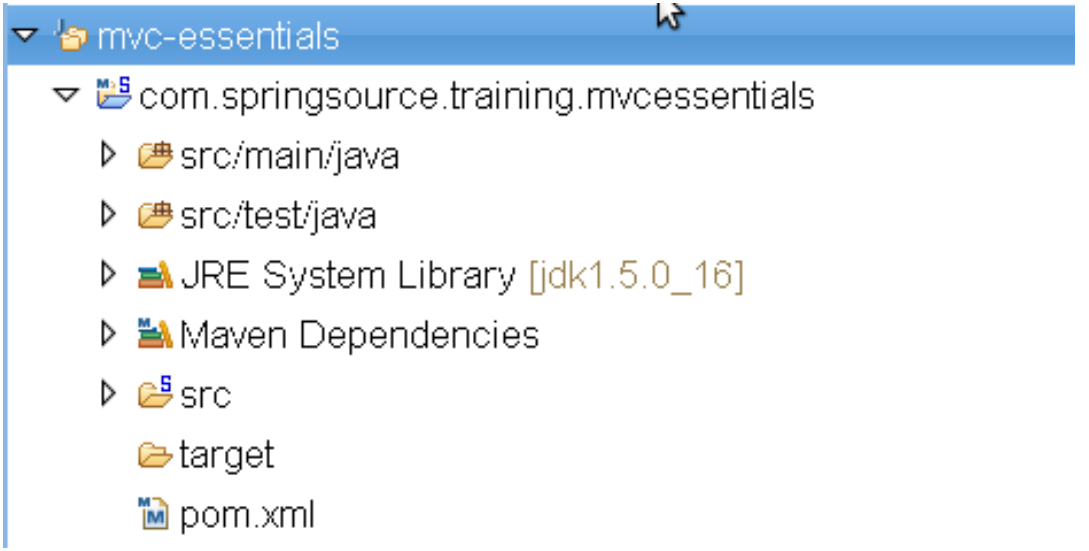
The Spring MVC DispatcherServlet is working, but it does not know how to handle your request for the /home resource. You will implement the welcome screen later. For now, lets review the existing code that is already

there for you.

2.2.2.2. Review project structure

In this step, you will become familiar with the overall structure of the mvc-essentials-1 project. This is the same structure that used for all of the lab projects.

Expand the mvc-essentials-1 project from within the Package Explorer of the Tool Suite:



Your source code resides under `src/main/java`. Your unit and integration tests reside under `src/test/java`. What about web content? Expand the `src` directory and navigate to `main/webapp`. This is the webapp root directory where your public web content resides, such as images and styles. Inside the `WEB-INF` subdirectory is where your protected web resources reside, such as `web.xml` and your JSP page templates.

2.2.2.3. Review web.xml

The configuration of every Java EE web application, including those powered by Spring, starts in `web.xml`. In this step, you'll learn how the web application is configured. Open `web.xml` and note the declaration of a Spring MVC `DispatcherServlet` with two initialization parameters.

Tip

If you prefer keyboard shortcuts to the Package Explorer, you can use CTRL+SHIFT+R to load any resource (non-Java) file. Be sure that you're opening the file from the correct project - you

may want to close projects that you're not actively working on (except for `00-reward-network`, `02-mvc-essentials-1` and `Other Projects`) to avoid confusion.

The contextConfiguration parameter specifies the Spring configuration files to use. In this application you'll use all the files inside `/WEB-INF/spring`.

Tip

Co-locating Spring configuration files in a directory like /WEB-INF/spring is a useful practice; it makes it easy to locate your configuration and make changes to it.

Note the servlet mapping used for the `DispatcherServlet`. The servlet handles all requests into the application and is mapped to the `/admin/*` path.

Tip

Mapping by logical path such as /admin/* is considered a best practice over mapping by extension (*.do, *.htm, etc).

Tip

In the previous section, we discussed eliminating the servlet element (i.e. /admin) in the URL. We will discuss how to do this following the lab. For now, just accept the servlet element as part of the URL.

2.2.2.4. Review Spring configuration

Spring manages the internal components of your application, and the Spring MVC Dispatcher Servlet handles mapping web requests to those components for processing. In this step, you'll review the Spring configuration that has been provided for you, and see what you will need to configure yourself.

Expand the `/src/main/webapp/WEB-INF/spring` directory. Notice there are two Spring configuration files: `app-config.xml` and `mvc-config.xml`. `app-config.xml` contains core application configuration not necessarily related to the web layer. `mvc-config.xml` contains specific configuration used to initialize Spring MVC. Most of the labs in this course use this configuration.

First, open `app-config.xml` and notice the component-scan directive at the top of the file. This directive tells Spring to scan the `rewardsonline` package and its subpackages for @Component classes to deploy as Spring

beans. Classes annotated with `@Controller`, `@Repository`, or `@Service` are all types of components. Spring will discover each class annotated with one of these annotations and automatically create and configure an instance of it!

Now review the existing @Components defined for this project. Navigate to your Java source in `src/main/java`. There you'll find the `rewardsonline` base package, and an `accounts` subpackage. Within accounts, open the `AccountManager` interface definition. Note this logical interface defines an operation to find all accounts (needed by the Account Listing page), and an operation to find a single account (needed by Account Details page).

Now select CTRL-T to view the AccountManager type hierarchy and note there are two implementations: a `HibernateAccountManager` used in production, and a `StubAccountManager` used in test. Open the HibernateAccountManager. Notice how it is annotated as a @Repository. Also note it depends on a Hibernate SessionFactory, which is passed into its constructor. Because this class is annotated as a @Repository, a type of @Component, it automatically gets discovered by Spring and deployed as a bean when the container initializes. The @Autowired hint on the constructor tells Spring to auto-inject a reference to the Hibernate SessionFactory managed by the container.

Tip

Hibernate mapping files showing how the Account entity maps to relational database structures are co-located with the HibernateAccountManager implementation. In general, packaging code by functional responsibility is considered a best practice, over packaging code by technical layer.

Tip

Each concrete implementation class in the project has an associated unit test located in `src/test/java`. Feel free to review the tests and even run them to verify they pass.

Now return to `app-config.xml` and review the remaining configuration there. You'll see the configuration for the Hibernate SessionFactory, as well as the DataSource that provides connectivity to the shared rewards database. There is also a transaction manager configured that starts transactions using Hibernate around @Transactional application methods.

Tip

It is a best practice to configure your own application components, such as an HibernateAccountManager, using annotations like @Repository and @Autowired. Configure infrastructure, such as your Hibernate SessionFactory and DataSource, using externalized XML configuration.

Lastly, right-click on `app-config.xml` in the Spring Explorer view and select Open Graph. This opens a visualization of the Spring Container as it will exist when your application starts. Notice how your TransactionManager is wired with your SessionFactory which is wired with your DataSource, illustrating the natural relationship between components.

Now quickly open the `mvc-config` containing the Spring MVC configuration. Notice it doesn't define any beans. You will be completing the MVC configuration as necessary to implement the Welcome, Account Listing, and Account Details use cases in the remaining sections. When you are ready to code, move on to the next section!

2.2.3. Implementing the Welcome Page

The simplest page you'll implement will be the welcome page, so you'll address it first. Recall when you access the root URL of the application, currently the redirect to the /home resource fails because the DispatcherServlet doesn't know about it. Confirm this again by accessing http://localhost:8080/mvcessentials-1 and verifying you see this entry in your logging output:

```
WARN : org.springframework.web.servlet.PageNotFound
No mapping found for HTTP request with URI [/mvcessentials-1/admin/home]
    in DispatcherServlet with name 'admin'
```

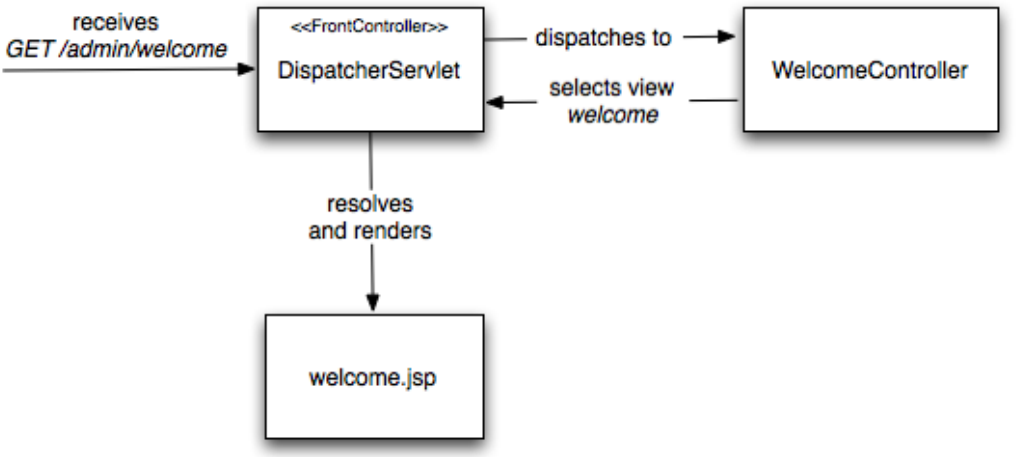
When you see a welcome page display successfully instead of this 404 NOT FOUND, you will have completed this section.

2.2.3.1. Devise technical approach

In this step, you'll devise a technical approach to implementing the welcome use case.

The template for the welcome page has already been authored for you and is located at `/WEB-INF/welcome.jsp`. Quickly open this file (Ctrl+Shift+R) and confirm its just a simple JSP that uses JSTL tags to render a welcome message. This is the JSP that should render when the welcome resource is accessed.

Now recall in a MVC architecture, all HTTP requests are first routed to Controllers for processing--requests never go directly to views. A Controller's job is to handle a request, *then* select the appropriate view to render the response. For a basic welcome page, there is no real request processing logic. You simply need to implement a Controller that selects the welcome view for rendering. The desire request processing pipeline is shown graphically below:



So it should be clear you need to implement a WelcomeController that renders your welcome page. Also, any one-time configuration required to get the DispatcherServlet request processing pipeline in-place should be added to `mvc-config.xml`. When you are comfortable with this technical approach, move on to the next step.

2.2.3.2. Create WelcomeController class

Now create your WelcomeController in the `rewardsonline` base package. Make it an annotated @Controller and define a single method to handle GET requests sent to the `/home` resource. Have your /home request handler select the `/WEB-INF/welcome.jsp` view to render. Keep your implementation very explicit for now; do not apply any conventions.

Tip

Recall: when a class is annotated as a `@Controller`, a type of `@Component`, no configuration is needed to deploy it as a bean. The one-line component-scan directive in `app-config` will find it!

After you have completed your WelcomeController implementation, give it a test drive. Try accessing http://localhost:8080/mvcessentials-1 again. You should see the welcome page render successfully! If you still get a 404, check your logging output; also consider placing a breakpoint on your WelcomeController method. Ask your instructor if you have questions.

When your welcome page is rendering successfully, move on to the next step!

2.2.3.3. Configure messages

You probably noticed your welcome view rendered message placeholders like ???welcome.title??? instead of the actual messages. In this step, you'll fix this and get your messages resolving properly.

Messages for the application reside in `/WEB-INF/messages/global.properties`. Quickly open this file (Ctrl+Shift+R) and scan what is there--you'll see messages organized by page and by domain object. To plug these messages into the DispatcherServlet's view rendering context, you simply need to configure a MessageSource bean. Spring MVC will automatically configure this bean to handle message lookups initiated by standard JSTL <fmt:message> tags in your JSP templates.

In `mvc-config.xml`, add a `ReloadableResourceBundleMessageSource` bean definition. Be sure to assign the bean an id of *messageSource*; this special id is required for the DispatcherServlet to detect it. Configure the bean with a property named basename with a value of `/WEB-INF/messages/global`.

Tip

Remember to use auto-complete (Ctrl+Space) for bean class names and property names to save on typing.

Tip

Note that tc Server does *not* automatically detect changes to Spring configuration files. Make sure you stop and restart the server after you change `mvc-config.xml`.

After your messageSource has been added, refresh the welcome page again and verify the messages now resolve successfully. Once your messages resolve, move on to the next step!

2.2.3.4. Optimize view resolution

Congratulations! Your welcome page renders successfully, handled nicely by your WelcomeController. However, one implementation detail is less than ideal--your @Controller returns a hardcoded path to /WEB-INF/welcome.jsp. This strong coupling between your Controller and a specific view template makes it more difficult to change the location of your views in the future. It also allows for views to reside outside of the protected /WEB-INF directory, which introduces a security risk. Finally, it makes it impossible for your Controller to render different representations of the same resource; for example, XML, PDF, or JSON instead of HTML. For all these reasons, it is a best practice to have your Controller return a *logical view name* instead of a physical path. Then, the calling DispatcherServlet can map that logical view name to a physical resource such as a JSP template. This keeps your Controllers simpler and provides more flexibility and security. You will do exactly that in this step.

Navigate to your WelcomeController. There, instead of returning the full path to the view resource, return the

logical view name *welcome*. The idea here is your Controller returns a simple logical name, then the DispatcherServlet handles mapping that name to a JSP inside /WEB-INF.

After making this basic change, refresh your welcome page. You should see a 404 NOT FOUND with the following in your log:

```
WARN : org.springframework.web.servlet.PageNotFound -
No mapping found for HTTP request with URI [/mvcessentials-1/admin/welcome]
    in DispatcherServlet with name 'admin'
```

This is because the DispatcherServlet tried to forward to the /welcome resource (it treated your welcome view name as a relative resource path). To fix this, return to your `mvc-config.xml` and configure the DispatcherServlet to map logical view names to protected .jsp templates inside your /WEB-INF/ directory. Do this by creating a `InternalResourceViewResolver` bean configured with the appropriate prefix and suffix, and then restarting the server.

With your ViewResolver defined, refresh your welcome page. It should render successfully again, now with your WelcomeController decoupled from the hardcoded JSP path. Move on to the next step!

2.2.4. Implementing the Account Listing Page

Congratulations! You have successfully implemented your first @Controller, as well as completed one-time Spring MVC setup. You should now have a good understanding of how to implement controllers, and how the DispatcherServlet request processing pipeline works. In this section, you will continue by authoring an AccountController to implement the two remaining use cases. The steps in this section will be lighter, giving you more freedom in implementation now that you have a solid Spring MVC foundation.

2.2.4.1. Devise technical approach

Open your /home page in your browser and access the Accounts link. You should receive a 404. Your task in this section is to implement a controller that handles this request. You'll know it's working when you see a listing of all accounts in your browser.

2.2.4.2. Create the AccountsController

Create your AccountsController in the `rewardsonline.accounts` package. Implement a method to handle GET /accounts requests. Within the body of this method, delegate to the AccountManager to find the list of accounts. Expose this list to the accounts/list view for rendering.

Tip

You can instruct Spring to inject a dependency such as an AccountManager by annotating a constructor argument, setter method, or field as @Autowired.

Tip

The view template has already been authored for you at `/WEB-INF/accounts/list.jsp`

As you implement, experiment with your @Controller method signature to learn the possible syntactical variations. First, have your method accept a Model argument and return a String. This is a common method signature that gives you full control over populating the Model and selecting the view.

Once the account listing displays successfully, move on to the next section. Take your time and experiment with the possible syntax variations to learn the ins-and-outs of the @Controller programming model. Ask your instructor if you have questions.

2.2.5. Implementing the Account Details Page

Complete this final section by implementing the use case to show account details. You can confirm this use case has not been implemented when you click on one of the accounts from the list page. You'll know you have it working when you see details displaying successfully.

To do this, simply add another method to your AccountsController to handle GET /accounts/{accountNumber} requests. Use the @PathVariable annotation to bind the segment of the URL that holds the accountNumber to a method argument. Delegate to the AccountManager to find the Account with that id. Expose the Account to the view for rendering. Use the @Controller method signature that feels most natural to you for this use case.

Tip

It is a best practice to group methods that act on the same logical resource, such as accounts, together in the same @Controller class.

Once you see the details of selected accounts, you have completed this section! Congratulations--you are well own your way to becoming a Spring MVC expert.

2.2.6. Unit Tests

2.2.6.1. Implement @Controller unit tests

@Controllers are easy to test because they often have few dependencies. In this step, create unit tests for your AccountsController and WelcomeController within the `src/test` tree. A stub AccountManager has been provided to help test your AccountsController. When all tests are passing, you have completed this section and the lab.

Chapter 3. mvc-layout: Using Tiles Layouts in Spring MVC

3.1. Introduction

In this lab, you will apply a common site layout to all existing pages.

What you will learn:

1. How to apply common layouts to your pages using Apache Tiles
2. How to configure Tiles for use with Spring MVC

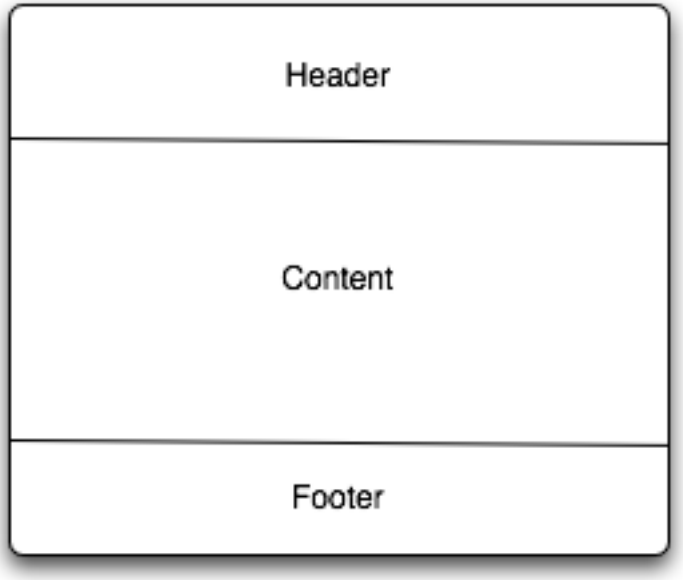
Estimated time to complete: 45 minutes

3.2. Instructions

The instructions for this lab will have you begin by factoring out common layout from `welcome.jsp` into a template. Next you will put together the Tiles configuration necessary to re-use the common layout for the `welcome.jsp` page.

3.2.1. Applying a page layout

The pages of most web applications share a common structure. A typical page structure consists of a header at the top, content in the middle, and a footer at the bottom:



Good templating systems allow you to define such a shared page structure in a single template called a layout. A layout is then applied to a page by inserting individual page elements into its structure. For any given page, the content usually varies while the header and footer elements usually remain the same.

Apache Tiles is a popular layout engine that works well with JSP templates. In this section, you will use Apache Tiles to apply a common layout to all of your pages.

3.2.1.1. Review technical approach

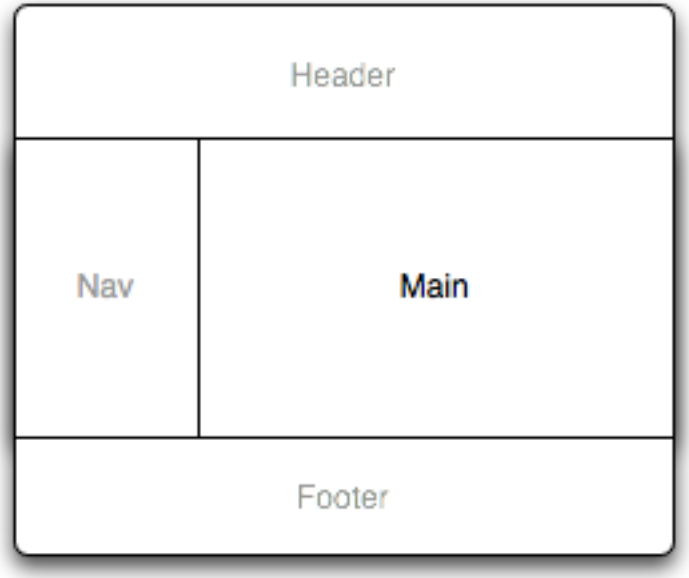
This lab picks up where the last lab left off. Confirm this by deploying the mvc-layout application to the server and accessing it at http://localhost:8080/mvclayout. You should see the familiar welcome screen.

Now select the Accounts link and notice the Account listing renders without a header, navigation, or footer. The same holds true for an account's details. This is because they now assume an overall layout is being applied. The welcome page, on the other hand, still embeds the layout with its content.

To complete this lab, you need to factor the layout information in the welcome page into a single layout that is applied to all three pages. You will know you are complete when the welcome, account listing, and account details pages all render with the correct layout, and there is a clear separation between your content and the page layout.

3.2.1.2. Factor out the layout information in welcome.jsp

First open `/WEB-INF/welcome.jsp` (CTRL+Shift+R) and review what is there. Notice there is a root "page" div defining the overall page structure. Within page are "header", "content", and "footer" sub elements. The content element is split into two sections, "main", for the main content area, and "nav", for the navigation menu. The body of the main element contains the content specific to this welcome page definition. This structure is shown graphically below:



In this step, you will factor out the shared page structure into a generic layout, leaving just the main content specific to this page.

Complete this step by first creating a `standard.jsp` in `/WEB-INF/layouts`. Cut the common page structure out of welcome.jsp and into the layout, leaving only the content of the . Have the main content body in your layout simply be empty for now. Verify you completed this step successfully by refreshing your welcome page and confirming all that renders is its content: a title and caption.

Tip

Keep in mind your welcome.jsp still needs to import the JSTL fmt tag library since it uses it to generate its main content.

3.2.1.3. Configure Tiles

You have successfully separated your page layout from its content. In this step, you will create Tiles definitions and configure Tiles for use with Spring MVC.

Start by creating a tiles definition file, `tiles.xml`, in your /WEB-INF directory. Paste in the following template to help you populate the file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE tiles-definitions PUBLIC
        "-//Apache Software Foundation//DTD Tiles Configuration 2.0//EN"
        "http://tiles.apache.org/dtds/tiles-config_2_0.dtd">

<tiles-definitions>

</tiles-definitions>
```

With the file created, go ahead and create a definition for your welcome page. The definition should extend from your standard layout.

After you create and save the Tiles definition, try refreshing your welcome page.

Tip

You'll need to restart the server whenever you make a change to your Tiles definitions, just like you do when you make Spring configuration modifications.

The layout is still not yet being applied! This is because Spring MVC is currently configured to render your JSP pages directly, rather than render Tiles definitions. Confirm this by inspecting your `mvc-config.xml` and reviewing the ViewResolver configuration. Notice you have a InternalResourceViewResolver configured that maps logical view names to JSP resources inside /WEB-INF. Instead of resolving JSPs directly, you need to map view names to tiles definitions. Tiles will then handle rendering these definitions, which define page compositions.

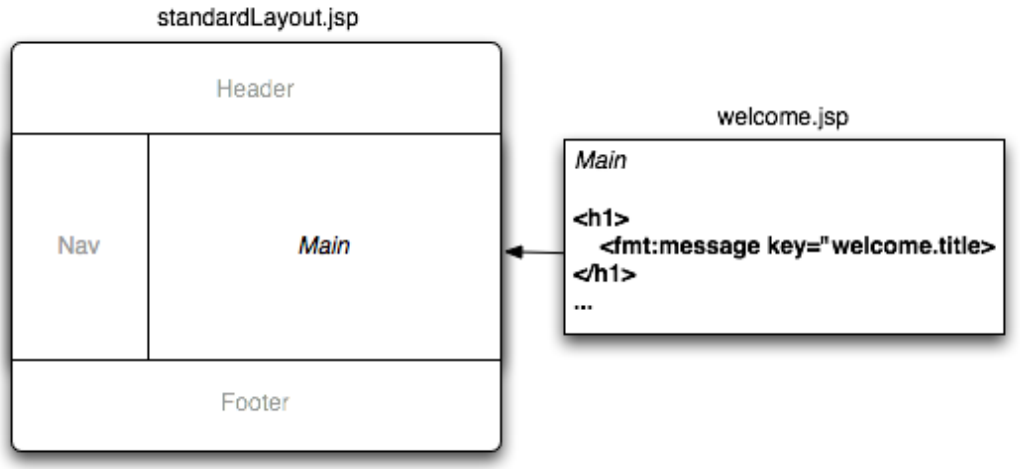
Plug in the Tiles rendering engine by swapping out the InternalViewResolver with a TilesViewResolver. Also recall that Tiles must be initialized before it can be used, so you will need to perform one more initialization step in your `mvc-config.xml`.

Now restart the server and try refreshing the welcome page. The page will render with the layout applied but without the main content (the Welcome heading and sub-heading). This confirms your welcome definition was successfully resolved by Tiles as part of the Spring MVC pipeline. The layout template however needs a little more work.

3.2.1.4. Complete the layout template

To complete the layout you will need to define the places where content may be inserted. You assign each of these places an unique name. Then, in your tiles definition you create a matching attribute that is a String or points to a JSP. Tiles does the rest.

For this lab, recall you should insert your page content into the "main" element of your layout. This is shown visually below:



Make this happen by opening `/WEB-INF/layouts/standard.jsp` (CTRL+Shift+R). Add a tiles taglib declaration (Ctrl+Space inside the uri attribute should help). Find the main content div, which should have an empty body at present. Use the <tiles:insertAttribute> tag to insert a page's `main` content at this spot. Finally, use the <tiles:putAttribute> tag to set the `main` content to be your welcome content.

Refresh the welcome page to test your changes. The page may fail with the error: "NoSuchAttributeException: Attribute 'main' not found." It means you have not defined a matching attribute in the tiles definition. Open `/WEB-INF/tiles.xml`, correct the problem, and restart the server.

Once the welcome page is rendering with the standard layout applied, do the same for the two remaining pages of the RewardsOnline application. You should not have to change the page templates or the layout to do this. When all pages are rendering with the layout applied, you're ready for the last step.

3.2.1.5. Add imported attributes

Inserted attributes and blocks (as shown above) work well in most cases. However, you may occasionally find

the need to provide request attributes that are controlled by your Tiles definitions. To do this, we can use the `tiles:importAttribute` tag in our JSPs.

In this step, we'll add logic to display which section of the site (accounts or homepage) the user is currently viewing. This will vary on a page-by-page basis, and is natural to be placed in the Tiles definitions.

Add a new attribute named `navigationTab` to each of your Tiles definition files. Provide one distinct value for the definitions in `/WEB-INF/tiles.xml` and another for the definitions in `/WEB-INF/accounts/tiles.xml`.

Now all that's left is to import the attribute into your layout. Use a `tiles:importAttribute` tag to import "navigationTab" into `standardLayout.jsp`. You can then use standard JSTL tags (`c:if` or `c:choose`) to apply tags to the appropriate section in the nav div. For example, if the attribute is "home", your JSP might look like:

```
<strong>
    <a href="<c:url value="/"/>">
        <fmt:message key="navigate.home"/>
    </a>
</strong>
```

Whereas if it's not, it might look like:

```
<a href="<c:url value="/"/>">
    <fmt:message key="navigate.home"/>
</a>
```

Apply this to each part of the nav div. Once each section of the menu renders properly, you're done!

Chapter 4. mvc-views: Views in Spring MVC

4.1. Introduction

In this lab, you will add support for downloading content in Microsoft Excel format.

What you will learn:

1. How to render data in Microsoft Excel format using Apache POI
2. How to configure Spring MVC for use with multiple view technologies

Estimated time to complete: 30 minutes

4.2. Instructions

The instructions for this lab are divided into several steps. Initially you see how to submit requests for different content types. Next you will use a `BeanNameViewResolver` to route requests to a custom Excel view. Lastly you will configure a `ContentNegotiatingViewResolver` for detecting content types. The "extra credit" section demonstrates returning JSON.

4.2.1. Return Accounts in Microsoft Excel format

Most web applications primarily generate HTML content, but for some use cases they also need to generate other content representations such as Adobe PDF or Microsoft Excel. In this section, you will enhance the RewardsOnline application to also be capable of rendering an Account listing as a Microsoft Excel spreadsheet.

Get started by accessing the welcome page and navigating to the Accounts page. From there, select the "Show as Excel" link. Notice selecting this link just reloads the HTML page; it should instead download an Excel document. You need to first understand what is going on.

First, notice that the "Show as Excel" URL ends in "/accounts.xls". In your browser's address bar, try removing the ".xls" extension or replacing it with another extension like ".html" or ".pdf". This makes no difference, and the HTML page reloads each time. In fact, all of these URLs are handled by the same method of `AccountsController` because its @RequestMapping rule ignores the extension. This is to your advantage because it is possible to reuse the AccountsController.list() method to load the same model data, while varying the view rendering technology. Basically, when /accounts is requested, the list should be rendered in the

default content type, HTML. When /accounts.xls is requested, the same list should be rendered in Microsoft Excel format. In essence, the client should use the file extension to request the desired representation of the resource. The same list() Controller method should execute for both representations; what varies is the view.

4.2.1.1. Plug in the AccountsExcelView

Open `AccountsExcelView` (Ctrl+Shift+T). This class extends from Spring MVC's `AbstractExcelView`, which uses the Apache POI library to generate Excel documents. Take a few moments to study the code. Notice how the accountList model is retrieved and the Apache POI API is used to generate the document. The base class takes care of writing the document to the HTTP response stream for you.

Fortunately, this class is already complete. All you need to do is to plug it in so when the "accounts.xls" view is selected, your AccountsExcelView is rendered. To do this, edit `mvc-config.xml` and configure a view resolver chain that involves a `BeanNameViewResolver`, a `TilesViewResolver`, and the `AccountsExcelView`.

You're done when the "Show as Excel" link returns a Microsoft Excel representation of the account list. If your computer has a program for displaying Excel documents you will be able to open the document. If not, you will be prompted to save the document to a file on the local disk.

Next remove the .xls extension in the URL and try loading the Accounts Listing page. You should get Excel content one more time. This because you probably had to configure the BeanNameViewResolver first in the order of priority so it always does the rendering. What you really want is the ability to alternate based on the file extension in the URL.

4.2.1.2. Configure the ContentNegotiatingViewResolver

Spring 3.0 introduced the `ContentNegotiatingViewResolver`, a view resolver which can delegate to other resolvers based on the type of content to be rendered. We'll use this resolver to render the view properly as either HTML or Excel content.

Open `mvc-config.xml` and add an instance of the `ContentNegotiatingViewResolver`. This will use file extensions by default to determine the type to render, which matches your goal. Add media type mappings for Excel and HTML and set the default content type of text/html. Also add a viewResolvers property and add set it to a list containing the bean resolvers configured in the previous step. The order is not important.

Once your `ContentNegotiatingViewResolver` has been configured, try the Accounts Listing page and the "Show as Excel" link again. This time you should get the right content for each request.

Tip

Note that if you type the URLs ending in "/accounts.html" and "/accounts.htm" directly into your browser, the HTML page will be rendered, using the `TilesView`. The "/accounts.html" URL is explicitly mapped to the "text/html" content type in the CNVR. The ".htm" extension

also is resolved correctly because internally the CNVR uses standard file type associations configured in the Java Activation Framework (JAF). As configured, the CNVR will fail to resolve a view for any other file extension. For example "/accounts.do" will result in an exception and display an HTTP Status 500 error page. As of Spring 3.0.3, you can disable JAF by setting the CNVR property `useJaf` to `true`. This will cause random file extensions to render HTML (the default content type) which is usually preferable.

Also, as of Spring 3.0.2, you can also set the CNVR property `useNotAcceptableStatusCode` to true. This will result in an HTTP response with a status code 406 if no view is found in which case the application can be configured to handle it appropriately (see also extra credit step below).

Congratulations! You have completed this lab.

4.2.2. EXTRA CREDIT: Return Accounts as JSON

This section shows how to return Account data in JavaScript Object Notation format (JSON). We will be using the MappingJacksonJsonView which always converts the content of the Model to JSON data using the Jackson object to JSON marshalling tool. This is one way to support REST using Spring MVC.

4.2.2.1. What happens when a format is not supported?

First of all navigate to any individual account and click on the "View as JSON" link. You should get an HTTP 500 error message.

In the servlet configuration file `mvc-config.xml`, modify the CNVR bean definition and set the `useNotAcceptableStatusCode` property to true. Restart the server and refresh the page. You should now get an HTTP 406 error instead. Now let's add JSON support.

4.2.2.2. Setting up the JSON View Resolver

This is exactly the same as in the notes and we have written it for you - it is `rewardsonline.accounts.JsonViewResolver`. Now we need to use it.

In the servlet configuration file `mvc-config.xml`, modify the CNVR bean definition to support JSON as a media type. If you look at the `MappingJacksonJsonView` you will see the content-type that it returns.

Now add the `JsonViewResolver` as another resolver.

4.2.2.3. Configuring the Jackson Mapper

The mapper tries its best to work out how to convert objects to and from JSON, but sometimes it needs help. When you make Spring create beans automatically you have to add annotations to your class so it knows which constructors and setters to use. Jackson has the same problem. If you refresh the accounts.json page again, you will get a Jackson error saying it doesn't know how to convert the Percentage class to JSON data.

We need to tell Jackson how to convert our `Percentage` class to JSON data. To do that we need to annotate the constructor that takes a `BigDecimal` with `@JsonCreator`. We also need to tell it which getter to use by annotating the `asBigDecimal` method with `@JsonValue`. Jackon now knows how to create a Percentage object and how to get the value inside it. If you are not sure what to do, look at `MonetaryAmount` as we have already done this class for you.

Refresh that web-page yet again and you should see account details in JSON format.

4.2.2.4. Hibernate Lazy Loading Errors

Return to the page that lists all accounts and click on the "View as JSON" link. You should get an exception with the message "failed to lazily initialize a collection of role: rewardsonline.accounts.Account.beneficiaries, no session or session was closed". You should also find a Hibernate `LazyInitializationException` exception in the stack-trace. Do you know why this is happening?

Lazy Loading in Hibernate

Hibernate uses a technique called lazy-loading which is essentially "fetch on demand". When the accounts were fetched, their beneficiaries were not. When we listed the accounts in an HTML page, we didn't use the beneficiary information, so it didn't matter. When we try to convert the list of accounts to JSON, the Jackson mapper does a deep conversion, trying to convert all the beneficiaries of each account to JSON too. Unfortunately they are not there.

Normally this doesn't matter. Hibernate replaces all references to related data (including collections of references) by a proxy. When you try to access the data for the first time, the proxy goes to the Hibernate session and asks it to load the related data. However if there is no session because the transaction has finished, a lazy-loading error occurs. Since most data is fetched via a transactional service (in our case the `HibernateAccounbtManager`, lazy-loading errors most commonly occur in the view-layer

To fix this problem quickly, we are going to force the `HibernateAccounbtManager` to also fetch the beneficiaries when it fetches the accounts. Open the class file and find the `findAllAccounts()` method. Modify the code to use the `eagerQuery` instead of the `accountQuery`. The "join fetch" clause forces the query to fetch the beneficiaries, as well as the accounts, all in the same query (the underlying SQL will involve a JOIN of the Account and Beneficiary tables).

Refresh the accounts.json page and you should now see them all as JSON format text.

Note

This solution should be USED CAREFULLY in a real system as it forces all the beneficiaries to be loaded every time an account is fetched, regardless of whether they are needed. There is no quick fix to this problem. Your options are:

1. Use a "join fetch" if the related objects are almost always needed.
2. Add a boolean parameter to the `findAllAccounts()` method - allowing beneficiary fetching to be requested, or not. The code runs the appropriate query.
3. Use the DTO pattern - return a Data Transfer Object that contains only the account data needed by the view layer. If the beneficiaries are needed, the act of copying them into the DTO will force the lazy fetch inside a transaction, so all is well. DTOs are tedious to write initially but in the long run thay allow your view layer to be independent of your domain objects. Thus if a domain object changes, the DTOs derived from it need not, limiting the effect of change in the system. This is often the best approach but it is not a "quick fix".

Chapter 5. mvc-forms: Building Forms in Spring MVC

5.1. Introduction

Most web applications use forms to process user input of some kind. In this lab, you will learn how to build forms with Spring MVC by implementing the Account Edit page. In the bonus section you can also implement the Account Search page.

Please *read all of each section first* - there are hints and tips after the instructions, which may go over the page/off the screen, and you may waste time struggling when the hint or tip would have helped.

What you will learn:

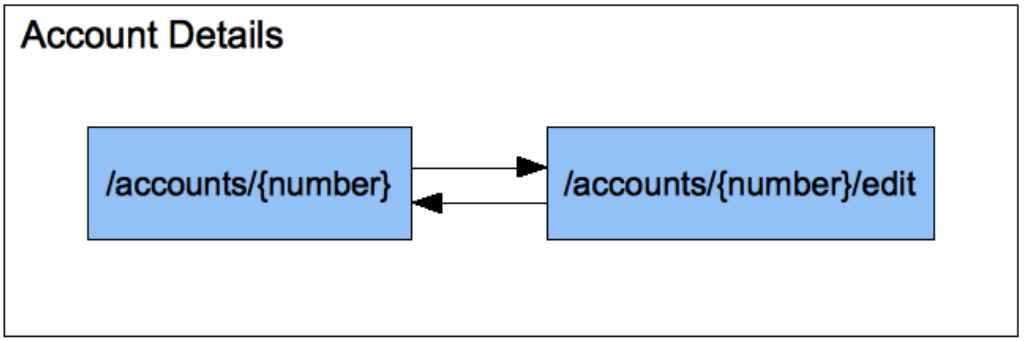
1. How to use the Spring Form Tag library to render forms populated with data.
2. How to apply formatting with formatting annotations.
3. How to work with data binding to have an object populated from form fields
4. How to invoke JSR-303 validation
5. How to process and display data binding and validation errors

Estimated time to complete: 45 minutes

5.2. Instructions

In this lab you will implement the Account Edit page.

Here is a graphic illustrating the desired page flow:



This is the HTTP contract desired for this functionality:

Table 5.1. HTTP Contract for RewardsOnline Account Processing

HTTP Method	Resource	Path Variables	Description
GET	/accounts/{number}	number: required, String	Get the Account with the given account number
GET	/accounts/{number}/edit	number: required, String	Get the form for editing the specified Account
POST	/accounts/{number}	number: required String	Update the specified Account

You can also view the page flow by deploying the mvc-forms-solution project and accessing it at http://localhost:8080/mvcforms-solution/accounts/123456789. The URL's in the solution should corresponds to the table and graphic above. When you have a good grasp of the requirements move on to the first section. Be careful not to edit the files in the solution by mistake - closing the mvc-forms-solution project is a good way to avoid this.

5.2.1. Implement The Account Edit Page

Deploy the `mvc-forms` project and access the Account Details page: http://localhost:8080/mvcforms/accounts/123456789. Or you can navigate to the account using the "List All Accounts" link on the home page - it is the first account in the list. The page should display the account data but the "Edit" link will not work. Confirm that you get a 404 error when you select "Edit".

5.2.1.1. Display The Account Edit Form

Open `AccountsController.java` and add a method to process requests for `/accounts/{number}/edit`. The method should be similar to the existing show() method except that it's based on the `accounts/edit` view. When this is done reload the edit page. You should now see a form although the form fields will be empty. Why?

Open `/WEB-INF/accounts/edit.jsp` (Ctrl+Shift+R) and examine the form. It's not at all surprising that the form shows empty fields, is it? Your next task is to bind the form fields to the properties of the Account model attribute.

Tip

Review the slides on using the Spring MVC form tag library and if you're still unsure ask the instructor.

Tip

Remember that using a custom tag library requires a taglib declaration. Fortunately code completion (Ctrl+Space) is available to help.

You're ready with this step when the form displays actual account data. The data may not be properly formatted just yet.

5.2.1.2. Add Form Field Formatting

Look closely at the `Date of Birth` field in the Account form. This uses the default format (e.g. "1981-04-11 00:00:00.0"). You'll need to format that according to application requirements instead. While JSTL format tags can be used to render the date, they'll be of no help when the form is submitted. You need a solution that correctly prints the date during rendering and parses it during form submission.

Open `Account.java`, and use the `@DateTimeFormat` annotation on the `dateOfBirth` field to specify the pattern to use, which is "yyyy-MM-dd". Both Spring MVC form tags (rendering) and the data binding mechanism (form submission) will be influenced by the @DateTimeFormat annotation.

Tip

Remember that date formatting is automatically enabled when using the custom MVC namespace `annotation-driven` element and the Joda Time library is available on the classpath.

Verify the formatting has been applied by refreshing the page in the browser.

5.2.1.3. Save Account Changes

In this step, you will process the form submit, update the account, and redirect to the account details page.

Add a method to process the form submission. Make sure the method is mapped to a request method of POST. Use the following guidelines to implement the method: accept an Account as input in order for data binding to take place; check if binding errors occurred and if so return to "accounts/edit"; if there were no binding errors, save the Account and redirect to the account details page.

Tip

A Controller may request a redirect by returning a view name that starts with `redirect:`. The text following `redirect:` specifies where to go next. If it begins with a slash it's relative to the web application context path. Otherwise it's relative to the current URL path.

Once you've created the method try submitting the form. You should see the following exception:

```
org.hibernate.TransientObjectException-The given object has a null identifier-rewardsonline.accounts.Account
```

This is Hibernate saying that it can't update the Account because it doesn't have its primary key set. Before reading on take a couple of minutes to see if you can figure out what's causing this. Have a look at the save method, think about how the Account was instantiated and populated.

To understand what the issue is start from the method in the `AccountController` class that processes the form submission. It accepts an `Account` instance, which Spring MVC will create and populate using form parameter values. But if an `Account` is created using it's default constructor will it have a primary key? The answer is "no". The `Account` must be retrieved from the database first and then updated using form parameter values. Your task is to make sure the `Account` passed to your method is retrieved from the database.

Tip

You might want to review the slides on Form Object Management first and if you're still unsure ask the instructor.

Tip

Once you've figured out how to ensure the `Account` is retrieved from the database, revisit the other methods and see if you can refactor to rely on the same mechanism.

You're ready to move on when the form saves correctly and redirects to the Account Details page.

5.2.1.4. Add Validation

In addition to data binding errors you may also need to apply additional validation rules. For example the Account `name` field is required, and so are `dataOfBirth` and `email`. Furthermore an email should comply to some regular expression (e.g. `"[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,4}"`). Add the appropriate JSR-303 annotations to the fields of the `Account` class.

Try editing the account again. Leave the name empty and save. Was the save processed without any error messages? Why?

Remember that JSR-303 validation is enabled during data binding with the help of the `@Valid` annotation on the input `Account`. Add that to the method that processes the form submit. This time validation errors should kick in and send you back to the edit page.

Tip

If you do end up on the edit page but no errors are displayed, make sure you have `form:errors` tags next to form input fields.

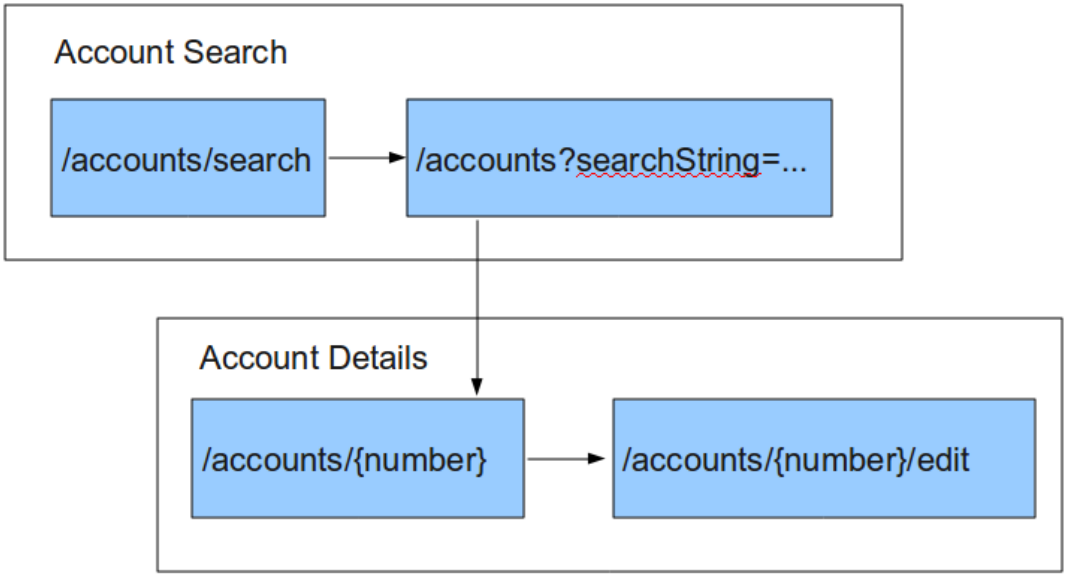
You've now completed this lab. If you'd like you can proceed with the bonus section.

5.2.2. Bonus: Implement The Account Search Page

As mentioned above, you have now completed the lab. However, you have the choice to continue with the following bonus material. Please note that you will probably need to do this outside normal class time unless you finish early.

By this point, you should have the ability to view and edit a selected account. However, how do you select an account in the first place?! In this section, you will add the capability to search for and select accounts.

The following illustration describes the full flow of the Account Search and Account Edit pages:



This includes several new HTTP resources, as defined in the following table:

Table 5.2. HTTP Contract for RewardsOnline Account Search Processing

HTTP Method	Resource	Request Parameters	Description
GET	/accounts/search	searchString (String), maxResults (int)	Gets the form for searching accounts. The form will be pre-populated with values from the last search.
GET	/accounts	searchString (required String), maxResults (int)	Get a list of accounts that meet the provided search criteria.

5.2.2.1. Display The Account Search Form

In this step, you will render the Account Search form. Start by opening the `AccountsSearchController` class in the `rewardsonline.accounts` package. Modify the existing method to process the request for a search form. Currently it returns the `notsupported` view, but now we need it to select the `accounts/search` view.

Tip

Recall it is a best practice to group control logic by logical *resource*. You can consider operations against sets of account resources, like a search, to be distinct from control operations against a single account resource, like show or edit, for example. Hence the use of the `AccountsSearchController` separately from `AccountsController`.

As with accounts, the Account Search form has already been created for you. Bind it to the `accountSearchCriteria` model attribute by adding the apporpriate Spring MVC form tags to `accounts/search.jsp`. Remember to add a corresponding object to the model in your controller method.

Tip

Note that this controller will be used for both new searches and to modify an existing search. Treating the AccountSearchCriteria as a form object with proper data binding will allow values from the last search to be displayed on the form.

Once you see the search criteria page, move on to the next and last step.

5.2.2.2. Implement the Account Search Results page

In this step, you will process search form submissions by displaying the results that match the user-entered criteria. First, open the search page's main template at `/WEB-INF/accounts/search.jsp` (CTRL+Shift+R). Note the resource/action the HTML form is configured to submit to, and the HTTP method that will be used. Also note the names of the two HTML input elements.

Now continue by authoring a new controller method to process the form submission. Bind the incoming form parameters to an instance of the `AccountSearchCriteria` class. Use the `AccountManager` dependency to find accounts meeting that criteria. Finally, expose the returned list of accounts in the model and select the `accounts/list` view to render the results.

Tip

The search form submits to a URL that we are already using to list all accounts - you can think of the search as a special case of listing accounts. However, you can't have two controller methods mapping to the same URL. To differentiate, for your new method, you will need to add an extra attribute to @RequestMapping to identify that this is a form submission. If you are still not sure, try submitting the form and look at the URL in the browser.

Tip

Remember the guidelines given in the presentation regarding search forms. If the returned list of accounts contains only one element, you should redirect to the accounts/{number} resource rather than rendering a list. However, there may be only account to present because we have reached the end of the list when paging - this is a different case and a list of one *should* be presented.

Tip

Don't forget to add validation annotations as appropriate to your AccountSearchCriteria object and controller method. Do *not* add @Valid to the search method you created in the previous step. If you do, you'll never be able to render the initial form!

Tip

Remember to convert the standard HTML for tags to the Spring form: tags.

Once this is complete, and you can successfully search, view, and edit accounts, you have completed this lab and bonus material! Congratulations!

Chapter 6. mvc-personalization: Enable Site Personalization Through Locale And Theme Switching

6.1. Introduction

In this lab you will give users the ability to change the look-and-feel of the site and also provide support for using a different language.

What you will learn:

1. How to use themes to control the look-and-feel of a website
2. How to configure locale switching in Spring MVC

Estimated time to complete: 30 minutes

6.2. Instructions

This instructions for this lab are divided in 2 parts. First you will experiment with switching themes and then do the same for locales.

6.2.1. Configure Theme Switching

Notice the link called "Green" at the top of the home page. If you click on it the page is reloaded with an extra query parameter "theme=green". Try reloading the page with "theme=blue". This should have caused a switch to the theme called blue but that doesn't work yet.

In your workspace find the `src/main/webapp/WEB-INF/classes` directory located under Web App Libraries (or use Ctrl+Shift+R). Here you'll find the property files for the blue and the green themes. Have a quick look inside. Now that you know the theme properties are in place you need to configure Spring MVC to use them.

6.2.1.1. Configure a ThemeChangeinterceptor

Open `mvc-config.xml` (Ctrl+Shift+R) and find the `mvc:interceptors` element. Interceptors defined here will be applied to all HandlerMapping instances in the application. Currently there is one interceptor for preventing

caching. Go ahead and add another interceptor of type `ThemeChangeInterceptor`.

When the server has redeployed the changes, try switching to the "blue" theme again. This time you will get an exception. Verify the exception occurred because the new interceptor detected the theme and tried to store it but failed to do so. This is because the default `FixedThemeResolver` does not support storing and hence switching themes.

6.2.1.2. Add a CookieThemeResolver

In `mvc-config.xml` add a `CookieThemeResolver` bean and set its "defaultThemeName" property to "green". Remember that the bean should have a specific id in order for it to be discovered by the DispatcherServlet.

By default, cookies created by the `CookieThemeResolver` will be *non-persistent* - they will be deleted as soon as the browser shuts down. To make cookies that outlast the browser, set the resolver's "cookieMaxAge" property to a value in seconds.

In a few moments the changes will be redeployed. If you now click the link multiple times you'll notice the theme parameter is alternating between "blue" and "green". Open `standard.jsp` (Ctrl+Shift+R) and verify the code that renders the link. You'll see that the code is inspecting the current theme to do the alternating. That means we're successfully switching themes even though there are no visual changes yet.

Another way to verify this is to inspect the cookie created by the `CookieThemeResolver`. In Firefox make sure the Web Developer toolbar is displayed (View - Toolbars - Web Developer). On the Web Developer toolbar find the menu called "Cookies" and then click the "View Cookie Information" item.

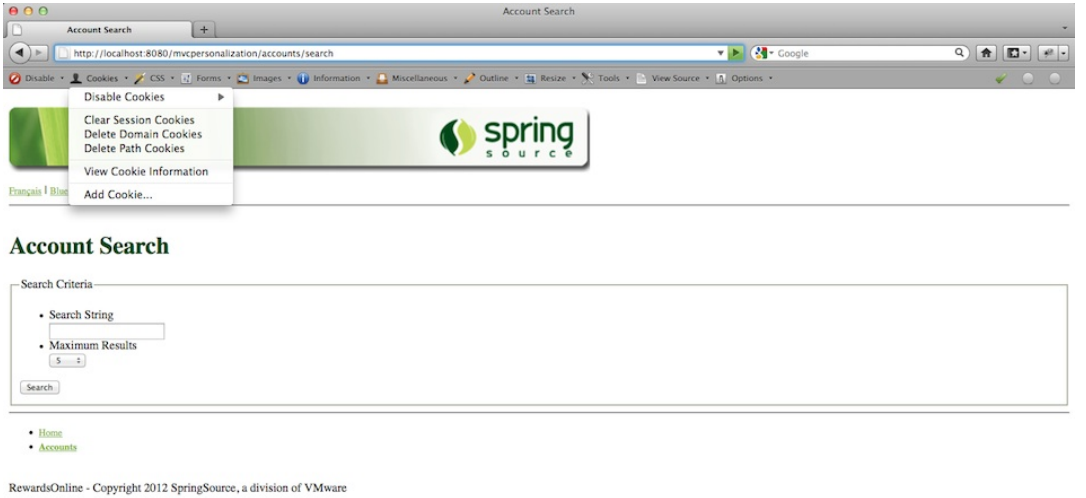


Figure 1: Viewing Cookie Information in Web Developer

If there are many cookies you can search (Ctrl+F) for "THEME" or otherwise just scroll down and verify the cookie value. Change the theme one more time and verify the cookie value has also changed.

All that remains now is to use the Spring theme tag to resolve and use theme properties.

6.2.1.3. Update standard.jsp To Use The Theme Tag

Open `standard.jsp` (Ctrl+Shift+R) and find the link to the "richweb-green.css" stylesheet. This is a theme-specific stylesheet and its location needs to be obtained through the Spring <theme> tag. Go ahead and do that now.

When done do the same for SpringSource banner image at the top of the page. Find the line that loads the "springsource_banner_green.png" image and make its path dynamic by using the theme tag.

When done try reloading the page. You should now be able to switch themes and see the visual changes take effect.

6.2.2. Add Locale Switching

The configuration required for changing locales is very similar to themes. Before starting out, open `web.xml` (Ctrl+Shift+R) and verify the presence of the `CharacterEncodingFilter`. This filter ensures the response is encoded as UTF-8 enabling the display of international characters.

6.2.2.1. Configure a LocaleChangeInterceptor

Go to `mvc-config.xml` and add a third interceptor of type `LocaleChangeInterceptor`. When the changes have published, navigate to the home page and attempt to change the locale by pressing the "Français" link. You will again see an exception because the default `AcceptHeaderLocaleResolver` does not allow storing locales other than the one that comes from the browser.

6.2.2.2. Configure a CookieLocaleResolver

In `mvc-config.xml` add a bean of type `CookieLocaleResolver` and set its "defaultLocale" property to "en". Remember that this bean must have a specific id for it to be discovered by the DispatcherServlet. When the changes have published test the language link at the top. You should be able to successfully switch between English and French.

Tip

If you want locale cookies to outlive the browser, the "cookieMaxAge" property can be used

here as well.

Once you can change the locale to French and back to English, you've completed this lab.

Chapter 7. rest-ws-1: Building RESTful Clients with Spring MVC

7.1. Introduction

In this lab you'll use some of the features that were added in Spring 3.0 to support RESTful clients.

What you will learn:

1. Writing a programmatic HTTP client to consume RESTful web services

Specific subjects you will gain experience with:

1. Using Spring's `RestTemplate`

Estimated time to complete: 30 minutes

7.2. Instructions

The RESTful service that you need has already been implemented. In this lab you will write a client application to access it. First you'll test retrieving existing data using a `RestTemplate`. Then you'll send requests to make changes to account beneficiaries.

7.2.1. Accessing accounts and beneficiaries as RESTful resources

In this section you'll access accounts and beneficiaries via RESTful resources using Spring's `RestTemplate` .

7.2.1.1. Inspect the current application

Under the `src/test/java` source folder you'll find an `AccountClientTests` JUnit test case: this is what you'll use to interact with the RESTful web services on the server. Your task will be to implement different tests in this class.

Firstly, deploy the application to your local server, start the server and verify that the application deployed successfully by accessing http://localhost:8080/rest-ws-1 from a browser. When you see the welcome page, the application was started successfully.

Now try to fetch all the accounts in JSON format by clicking on the `[JSON]` link on the welcome page. You should get a popup containing a scrollable list of all the accounts using JSON representation (JavaScript Object Notation). If so, the server is working properly and the REST service is working.

Hover over the `JSON` link with your mouse and you will see the URL being passed to fetch the accounts - you will need this int the next section.

7.2.1.2. Quick Instructions

If you feel you have a good grasp of how REST and Spring's `RestTemplate` work, all you need to do is open `AccountClientTests` and implement all the TODOs in turn. The `[JSON]` links in the web-application may help you.

Otherwise, detailed, step by step instructions follow.

7.2.1.3. Retrieve a list of accounts using a `RestTemplate`

A client can process the JSON contents anyway it sees fit. In our case, we'll rely on an HTTP Message Converter to deserialize the JSON contents into `Account` objects. Open the `AccountClientTests` class under the `src/test/java` source folder in the `accounts.client` package. This class uses a plain `RestTemplate` to connect to the server. Use the supplied template to retrieve the list of accounts from the server, from the same URL that you used in your browser (`TODO 01`).

Tip

You can use the `BASE_URL` variable to come up with the full URL to use.

Note

We cannot assign to a `List<Account>` here, since Jackson won't be able to determine the generic type to deserialize to in that case: therefore we use an `Account[]` instead.

When you've completed this `TODO` , run the test and make sure that the `listAccounts` test succeeds. You'll make the other test methods pass in the following steps.

7.2.1.4. Expose a single account

To expose a single account, we'll use the same `/accounts` URL followed by the `number` of the `Account` , e.g. `/accounts/123456789` .

Note

If you go to http://localhost:8080/rest-ws-1/accounts each account in the list has two links. If you click on its account number you will get the details as an HTML page. If you click on the `[JSON]` link you will get the same data as JSON. The same URL is invoked each time - but the Accept header in the request is different. (If you are interested, check the JavaScript in `src/main/webapp/WEB-INF/layouts/standard.jsp` to see how the JSON call works.)

One of the nice features of RESTful interfaces is that you can invoke them directly from a standard browser. Unlike SOAP web-services for example.

Complete `TODO 02` in the `AccountClientTests` by retrieving the account with number 123456789.

Tip

The `RestTemplate` also supports URI templates, so use one and pass "123456789" as the value for the `urlVariables` varargs parameter.

Run the test and ensure that the `getAccount` test now succeeds as well.

7.2.1.5. Create a new account

So far we've only exposed resources by responding to GET methods: now we'll try creating a new account as a new resource.

Complete `TODO 03` by POSTing the given `Account` to the `/accounts` URL. The `RestTemplate` has two methods for this: use the one that returns the location of the newly created resource and assign that to a variable. Then complete `TODO 04` by retrieving the new account on the given location. The returned `Account` will be equal to the one you POSTed, but will also have received a new account `number` when it was saved to the database.

Run the tests again and see if the `createAccount` test runs successfully. Regardless of whether this is the case or not, proceed with the next step!

7.2.1.6. Seeing what happens at the HTTP level

If your test did not work, you may be wondering what caused an error. Because of all the help that you get from Spring, it's actually not that easy to see what's happening at the HTTP transport level in terms of requests and responses when you exercise the application.

For debugging or monitoring HTTP traffic, Eclipse ships with a built-in tool that can be of great value: the TCP/IP Monitor. To open this tool, which is just an Eclipse View, press Ctrl+3 and type 'tcp' in the resulting

popup window; then press Enter to open the TCP/IP Monitor View. Click the small arrow pointing downwards and choose "properties".

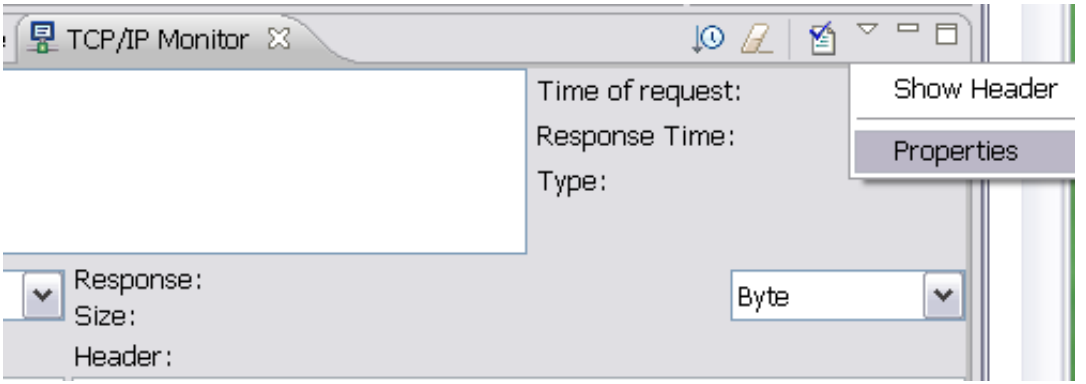


Figure 1: The "properties" menu entry of the TCP/IP Monitor view

Choose "Add..." to add a new monitor. As local monitoring port, enter 8081 since this port is probably unused. As host name, enter "localhost" and as port enter 8080 since this is the port that Tomcat is running on. Press OK and then press "Start" to start the newly defined monitor.

Tip

Don't forget to start the monitor after adding it! The most common error at this point is to forget to start the monitor.

Now switch to the `AccountClientTests` and change the `BASE_URL` 's port number to 8081 so all requests pass through the monitor.

Note

This assumes that you've used that variable to construct all your URLs: if that's not the case, then make sure to update the other places in your code that contain the port number as well!

Now run the tests again and switch back to the TCP/IP Monitor View (double-click on the tab's title to maximize it if it's too small). You'll see your requests and corresponding responses. Click on the small menu arrow again and now choose 'Show Header': this will also show you the HTTP headers, including the Location header you speficied for the response to the POST that created a new account.

Note

Actually, there's one request missing: the request to retrieve the new account. This is because the monitor rewrites the request to use port 8080, which means the Location header will include that port number instead of the 8081 the original request was made to. We won't try to fix that in this lab, but it wouldn't be too hard to come up with some interceptor that changes the port number to make all requests pass through the filter.

If your `createAccount` test method didn't work yet, then use the monitor to debug it. Proceed to the next step when the test runs successfully.

Tip

If your server is not working (for example producing a 500 error), try looking at its output. Click on the console tab, then click on the console selection icon (the monitor symbol) on the right hand side and select the monitor for Apache Tomcat. You will probably see a stack-trace from the server.

7.2.1.7. Create and delete a beneficiary

Still in `AccountClientTests` , complete `TODO`s 5 to 8. When you're done, run the test and verify that all test methods run successfully. If so, you've completed the lab!

Chapter 8. rest-ws-2: Building a REST Server with Spring MVC

8.1. Introduction

In this lab you'll use some of the features that were added in Spring 3.0 to build a RESTful web service. Note that there's more than we can cover in this lab, please refer back to the presentation for a good overview.

What you will learn:

1. Working with RESTful URLs that expose resources
2. Mapping request- and response-bodies using HTTP message converters

Specific subjects you will gain experience with:

1. Processing URI Templates using `@PathVariable`
2. Using `@RequestBody` and `@ResponseBody`
3. Using `@RequestStatus` for normal and exceptional responses

Estimated time to complete: 40 minutes

8.2. Instructions

The instructions for this lab are organized into sections. In the first section you'll add support for retrieving a JSON-representation of accounts and their beneficiaries and test that using the `RestTemplate`. In the second section you'll add support for making changes by adding an account and adding and removing a beneficiary. The optional bonus section will let you map an existing exception to a specific HTTP status code.

Note

Please note:

1. These notes do not assume that you have completed `rest-ws-1`. If you did do that lab, you will have built the `AccountClientTests` used by this lab. Now we are going to write the server to go with it.

2. There is some overlap in the instructions (how the converters work and using the HTTP Monitor) but at no point do you write the same code twice. Please follow the instructions or you may miss a step!

3. The web-application contains links that allow you to make RESTful JSON calls from your browser (these were also available in the `rest-ws-1` lab. These links don't work yet, but they will allow you to easily test your code from your browser (and use browser tools like Firebug) as an alternative to using the `AccountClientTests` JUnit test.

8.2.1. Exposing accounts and beneficiaries as RESTful resources

In this section you'll expose accounts and beneficiaries as RESTful resources using Spring's URI template support, HTTP Message Converters and test using a `RestTemplate`.

8.2.1.1. Inspect the current application

Look at the classes defined in the `rewardsonline.accounts` package. The MVC functionality has been split across *two* controllers:

1. The `AccountSearchController` contains the code for account searching (see the `mvc-forms` lab). All requests to `/accounts` map to this controller - fetching all accounts, or searching for some of them.

2. The `AccountsController` only handles activities relating to individual, existing accounts. All requests to `/accounts/{number}` map to this controller - getting and modifying accounts and managing beneficiaries

The `src/main/webapp/WEB-INF/spring/mvc-config.xml` contains the configuration for Spring MVC. The `<mvc:annotation-driven/>` element ensures that a number of default HTTP Message Converters will be defined (including one to handle JSON format data) and that we can use the `@RequestBody` and `@ResponseBody` annotations in our controller methods.

Under the `src/test/java` source folder you'll find an `AccountClientTests` JUnit test case: this is what you'll use to interact with the RESTful web services on the server. This class has been written for you (it's the solution to the `rest-ws-1` lab). When you server is correctly configured and running, this unit test should be able to run successfully.

8.2.1.2. Expose the list of accounts

Open the `AccountSearchController`. It already contains a method to fetch all accounts in HTML format, `list`, plus methods for account searching. At the bottom is a section marked `REST Methods`.

Complete `TODO 01` by adding the necessary annotations to the `listData` method to make it respond to GET requests to `/accounts`.

Tip

You need one annotation to map the method to the correct URL and HTTP Method, and some way to define when this method is used instead of `list`. Note that the controller is already annotated at class level with `@RequestMapping(value="/accounts")`.

Use another annotation to ensure that the result will be written to the HTTP response by an HTTP Message Converter (instead of an MVC View).

When you've done that, deploy the application to your local server, start the server and verify that the application deployed successfully by accessing http://localhost:8080/rest-ws-2 from a browser. When you see the welcome page, the application was started successfully.

First try to fetch all the accounts by clicking on `List All Accounts`. You haven't touched this method, but if it doesn't run it means you have broken the controller and it isn't running - check the Tomcat output on the console. The most likely error is that your new method conflicts with the existing `list` - you have mapped them both to the same URL. Fix this error if necessary.

Return to the home page and fetch the accounts in JSON format by clicking on the `[JSON]` link on the welcome page. You should get a popup containing a scrollable list of all the accounts using JSON representation (JavaScript Object Notation). If so, the server is working properly and your REST service is working.

But how is it possible that JSON data has 'magically' appeared?

The reason is that the project includes the Jackson library on its classpath:

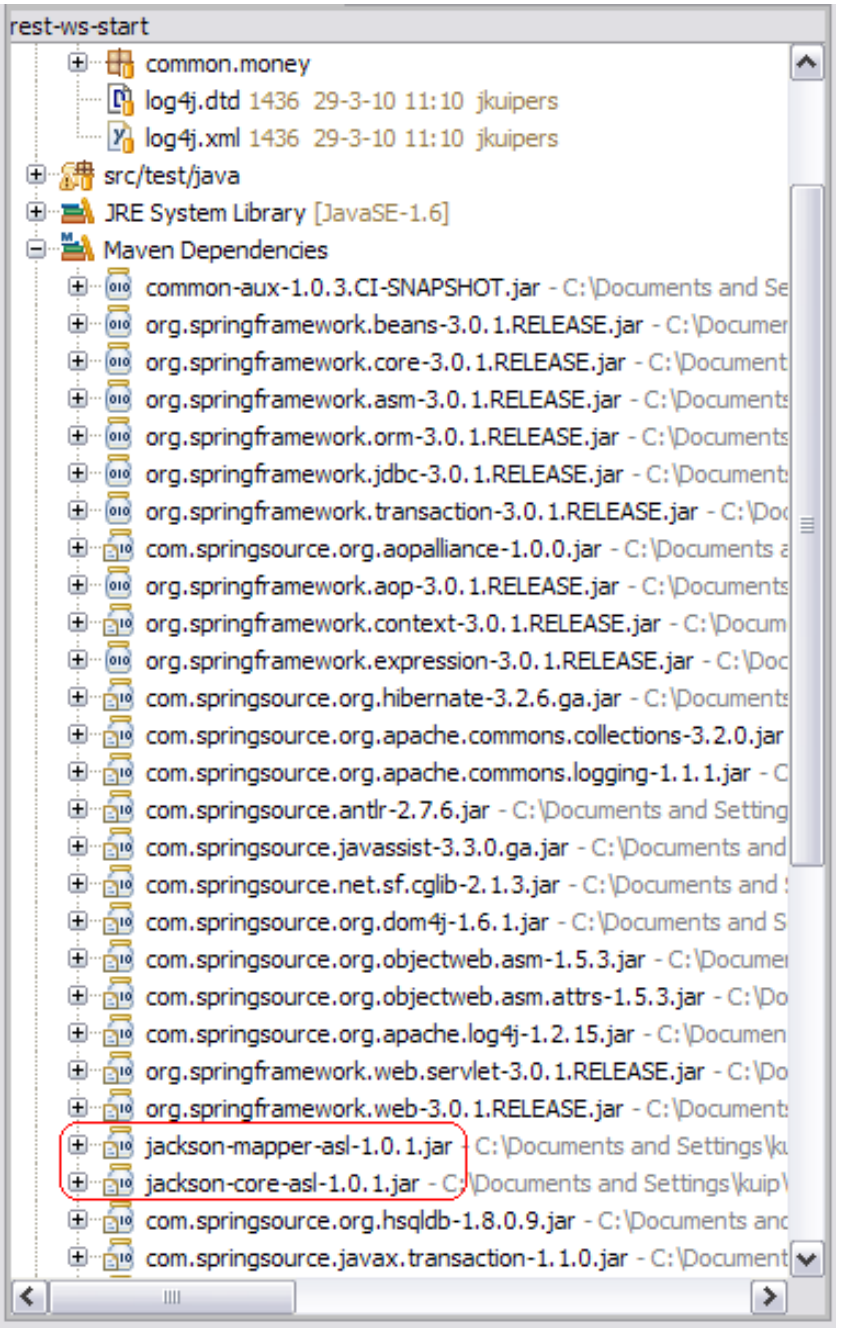

rest-ws-start
common.money
log4j.dtd 1436 29-3-10 11:10 jkuipers
log4j.xml 1436 29-3-10 11:10 jkuipers
src/test/java
JRE System Library [JavaSE-1.6]
Maven Dependencies
common-aux-1.0.3.CI-SNAPSHOT.jar - C:\Documents and Se
org.springframework.beans-3.0.1.RELEASE.jar - C:\Documer
org.springframework.core-3.0.1.RELEASE.jar - C:\Document
org.springframework.asm-3.0.1.RELEASE.jar - C:\Documents
org.springframework.orm-3.0.1.RELEASE.jar - C:\Documents
org.springframework.jdbc-3.0.1.RELEASE.jar - C:\Document
org.springframework.transaction-3.0.1.RELEASE.jar - C:\Doc
com.springsource.org.aopalliance-1.0.0.jar - C:\Documents a
org.springframework.aop-3.0.1.RELEASE.jar - C:\Documents
org.springframework.context-3.0.1.RELEASE.jar - C:\Docum
org.springframework.expression-3.0.1.RELEASE.jar - C:\Doc
com.springsource.org.hibernate-3.2.6.ga.jar - C:\Documents
com.springsource.org.apache.commons.collections-3.2.0.jar
com.springsource.org.apache.commons.logging-1.1.1.jar - C
com.springsource.antlr-2.7.6.jar - C:\Documents and Setting
com.springsource.javassist-3.3.0.ga.jar - C:\Documents and
com.springsource.net.sf.cglib-2.1.3.jar - C:\Documents and
com.springsource.org.dom4j-1.6.1.jar - C:\Documents and S
com.springsource.org.objectweb.asm-1.5.3.jar - C:\Docume
com.springsource.org.objectweb.asm.attrs-1.5.3.jar - C:\Do
com.springsource.org.apache.log4j-1.2.15.jar - C:\Documen
org.springframework.web.servlet-3.0.1.RELEASE.jar - C:\Do
org.springframework.web-3.0.1.RELEASE.jar - C:\Document
jackson-mapper-asl-1.0.1.jar - C:\Documents and Settings\ku
jackson-core-asl-1.0.1.jar - C:\Documents and Settings\kuip
com.springsource.org.hsqldb-1.8.0.9.jar - C:\Documents and
com.springsource.javax.transaction-1.1.0.jar - C:\Document

Figure 1: The Jackson library is on the classpath

If this is the case, an HTTP Message Converter that uses Jackson will be active by default when you specify `<mvc:annotation-driven/>`. The library mostly 'just works with our classes without further configuration: if you're interested you can have a look at the `MonetaryAmount` and `Percentage` classes in package `common` and search for the `@JsonXXXX` annotations to see the additional configuration.

8.2.1.3. Retrieve the list of accounts using a `RestTemplate`

A client can process the shown JSON contents anyway it sees fit. In our case, we'll rely on the same HTTP Message Converter to deserialize the JSON contents back into `Account` objects. Open the `AccountClientTests` class under the `src/test/java` source folder in the `accounts.client` package. This class uses a plain `RestTemplate` to connect to the server. Run the test and make sure that the `listAccounts` test succeeds. You'll make the other test methods pass in the following steps.

8.2.1.4. Expose a single account

To expose a single account, we'll use the same `/accounts` URL followed by the `accountNumber` of the `Account`, e.g. `/accounts/123456789`. Switch to the `AccountsController` and complete `TODO 02` by completing the `accountDetails` method.

Tip

This controller is already annotated at class level with `@RequestMapping(value="/accounts/{accountNumber}")`. Since the `{accountNumber}` part of the URL is variable, use the `@PathVariable` annotation to extract its value from the URI template that you use to map the method to GET requests to the given URL.

To test your code, click on `List All Accounts` on the welcome page. This will generate a list of all accounts. Each has a `[JSON]` link next to it - click on one and you should get a popup window showing the account details in JSON format.

Finally run the `AccountClientTests` and ensure that the `getAccount` test now succeeds as well.

8.2.1.5. Create a new account

So far we've only exposed resources by responding to GET methods: now you'll add support for creating a new account as a new resource.

Switch back to the `AccountSearchController` and implement `TODO 03` by making sure the `createAccount` method is mapped to POSTs to `/accounts`. The body of the POST will contain a JSON representation of an `Account`, just like the representation that our client received in the previous step: make sure to annotate the `account` method parameter appropriately to let the request's body be deserialized! When the method completes

successfully, the client should receive a `201 Created` instead of `200 OK`, so annotate the method to make that happen as well.

RESTful clients that receive a `201 Created` response will expect a `Location` header in the response containing the URL of the newly created resource. Complete `TODO 03` by setting that header on the response.

Tip

To help you coming up with the full URL on which the new account can be accessed, we've provided you with a static helper method on `AccountController` called `getLocationForChildResource`. Since URLs of newly created resources are usually relative to the URL that was POSTed to, you only need to pass in the original request and the identifier of the new child resource that's used in the URL and the method will return the full URL, applying URL escaping if needed. This way you don't need to hard-code things like the server name and servlet mapping used in the URL in your controller code!

Tip

Do not use `HttpServletRequest` or `HttpServletResponse` in your method. Instead use @Value and HttpEntity to generate and set the `Location` header.

When you're done, run the `AccountClientTests` tests again and see if the `createAccount` test runs successfully. Regardless of whether this is the case or not, proceed with the next step!

8.2.1.6. Seeing what happens at the HTTP level

If your test did not work, you may be wondering what caused an error. Because of all the help that you get from Spring, it's actually not that easy to see what's happening at the HTTP transport level in terms of requests and responses when you exercise the application. For debugging or monitoring HTTP traffic, Eclipse ships with a built-in tool that can be of great value: the TCP/IP Monitor. To open this tool, which is just an Eclipse View, press Ctrl+3 and type 'tcp' in the resulting popup window; then press Enter to open the TCP/IP Monitor View. Click the small arrow pointing downwards and choose "properties".

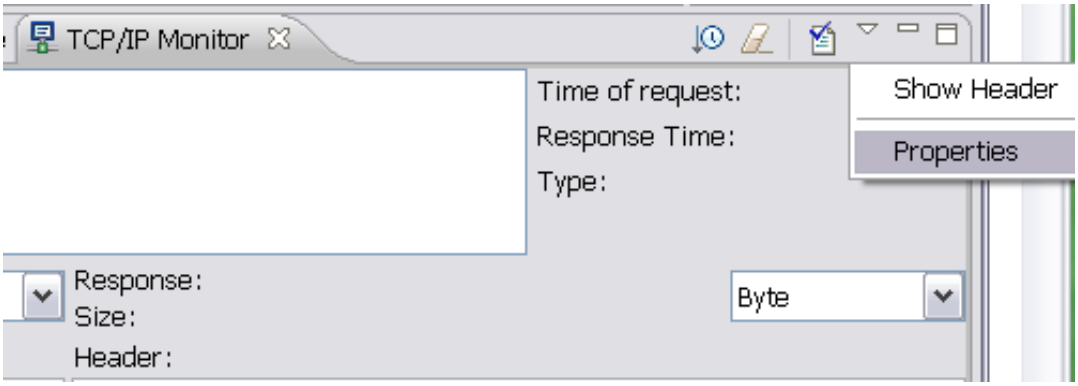


Figure 2: The "properties" menu entry of the TCP/IP Monitor view

Choose "Add..." to add a new monitor. As local monitoring port, enter 8081 since this port is probably unused. As host name, enter "localhost" and as port enter 8080 since this is the port that Tomcat is running on. Press OK and then press "Start" to start the newly defined monitor.

Tip

Don't forget to start the monitor after adding it! This is the most common error at this point.

Now switch to the `AccountClientTests` and change the `BASE_URL`'s port number to 8081 so all requests pass through the monitor.

Note

This assumes that you've used that variable to construct all your URLs: if that's not the case, then make sure to update the other places in your code that contain the port number as well!

Now run the tests again and switch back to the TCP/IP Monitor View (double-click on the tab's title to maximize it if it's too small). You'll see your requests and corresponding responses. Click on the small menu arrow again and now choose 'Show Header': this will also show you the HTTP headers, including the Location header you speficied for the response to the POST that created a new account.

Note

Actually, there's one request missing: the request to retrieve the new account. This is because the monitor rewrites the request to use port 8080, which means the Location header will include that port number instead of the 8081 the original request was made to. We won't try to fix that in

this lab, but it wouldn't be too hard to come up with some interceptor that changes the port number to make all requests pass through the filter.

If your `createAccount` test method didn't work yet, then use the monitor to debug it. Proceed to the next step when the test runs successfully.

8.2.1.7. Create and delete a beneficiary

Back to the `AccountController` class. Complete `TODO 04` by completing the `addBeneficiary` method in the `AccountController`. This is similar to what you did in the previous step, but now you also have to use a URI template to parse the `accountNumber`. Make sure to return a `201 Created` status again! This time, the request body will only contain the name of the beneficiary: an HTTP Message Converter that will convert this to a `String` is enabled by default, so simply annotate the method parameter again to obtain the name.

Finish `TODO 04` by setting the Location header to the URL of the new beneficiary.

Note

As you can see in the `getBeneficiary` method just below, that the name of the beneficiary is used to identify it in the URL. This is the `childIdentifier` that you need.

Finally follow `TODO 05` and complete the `removeBeneficiary` method. This time, return a `204 No Content` status.

To test your work, switch to the `AccountClientTests`, run the test and verify that this time all test methods run successfully. If this is the case, you've completed the lab!

8.2.1.8. BONUS 1 (Optional): return a 404 Not Found fetching an unknown account

At the end of the last test, we deleted a beneficiary, but we don't actually know for certain whether it worked. We need to try and fetch it and prove it's no longer there.

The correct response to a missing resource is the familiar HTTP 404 Status Code. Implement `TODO 06` by adding an exception handler to the `AccountController`. The `handleNotFound` is provided for you to start from.

In the `AccountClientTests` there is also a TODO 06 - enable the `checkDeleted` variable so the check will occur. Rerun the `AccountClientTests` and all tests should still pass.

8.2.1.9. BONUS 2 (Optional): return a 409 Conflict for duplicate account number

The `createAccount` test ensures that we always create a new account using a (probably) unique random number. Let's change that and see what happens. Edit the `optionalCreateExistingAcount` method in the test case and enable the `optionalTest` variable so it runs - see `TODO 07`.

Run the test and it should fail. When you look at the exception in the JUnit View or at the response in the TCP/IP monitor, you'll see that the server returned a `500 Internal Server Error`. If you look in the Console View for the server, you'll see what caused this: a specific subclass of `SQLException` - a `SQLIntegrityConstraintViolationException` - indicating that the number is violating a uniqueness constraint.

This isn't really a server error: this is caused by the client providing us with conflicting data when attempting to create a new account. To properly indicate that to the client, we should return a `409 Conflict` rather than the `500 Internal Server Error` that's returned by default for uncaught exceptions.

To make it so, complete `TODO 07` in `AccountSearchController` by adding a new exception handler for this situation. This exception will be wrapped either by Spring as a `DataIntegrityViolationConstraint` exception or by Hibernate as a `ConstraintViolationException`. Make your handler catch both, just to be safe.

Warning

Don't pick up `javax.validation.ConstraintViolationException` by mistake

When you're done, run the test again and check that the last test now runs because we are getting the expected status code.

Chapter 9. ajax: Building an AJAX Search

9.1. Introduction

Many web applications use forms to process user input of some kind. In a prior lab you performed a typical 'edit' scenario where a form was used to capture changes and then POSTed back to the server. You may have optionally implemented the search scenario by also submitting a request to the server using a GET type invocation and then were re-directed to a new page with the search results. In this lab, you will put into practice some of the AJAX principles in order to build a more rich style of interaction for performing the search.

What you will have a chance to practice:

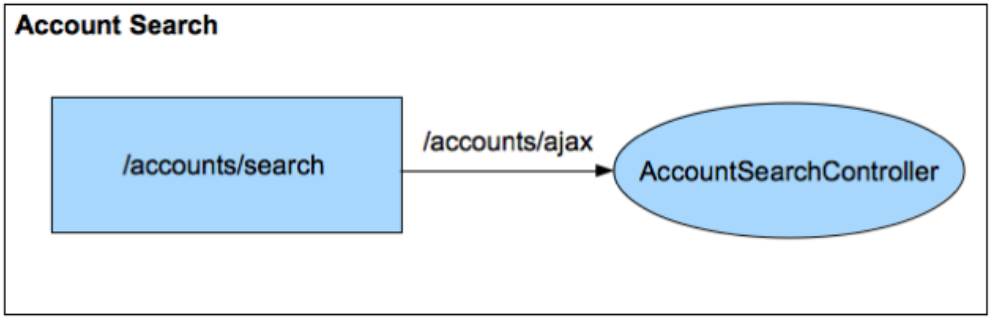
1. How to use the JQuery library to decorate and modify the HTML elements on the page
2. How to formulate an AJAX request using the JQuery library
3. How to handle the AJAX response by dynymaically updating the page with the response data
4. How to configure the`MappingJacksonJsonView` and the `ContentNegotiatingViewResolver` to re-use existing controller functionality for AJAX requests
5. How to use annotations to help customize the JSON mapping behavior

Estimated time to complete: 45 minutes

9.2. Instructions

In this lab you will modify the Account Search feature to use the AJAX style of invocation.

Here is a graphic illustrating the desired page flow:



Note that the use of the path `/ajax` is somewhat arbitrary. This path is used simply to distinguish the AJAX account search request from the existing search request.

9.2.1. Implement JQuery and AJAX functionality to perform a filtered search

Start by deploying the `ajax-start` project and accessing it at http://localhost:8080/ajax-start/.

9.2.1.1. Assess the current implementation

Select the `Accounts` link and notice that under the search criteria form there are two labels; `Account` and `Name`. Open the `accounts/search.jsp` file under the `WEB-INF` directory and scroll down to the bottom of the file. Notice there is the start of a table definition. This is where the search results will ultimately be displayed.

Next, open the `layouts/standard.jsp` file and notice there is an entry to load the JQuery javascript library. This means that JQuery Javascript functionality is available on all pages that use this template layout file.

9.2.1.2. Use JQuery to initially hide the table

As a first step, you will want to initially hide the table. Create a JQuery document ready handler function to hide the table. Later, you will use this handler function to also intercept the search button click event and route to a JavaScript method.

Tip

Review the slides on using JQuery and if you're still unsure ask the instructor.

In the handler function, use the appropriate JQuery selector to select the `div` block that wraps the table and then use the hide method. Save the JSP and refresh the page.

Verify that the table header is indeed now hidden.

9.2.1.3. Attach a button clicked handler to the search button

In this step, you will add an event handler to intercept the submit button click event. You will direct the click event to a Javascript function that will process the form data and send via AJAX request to a request handler method on the `AccountSearchController` class.

First, bind an event handler function for the click event on the `Search` button in the document ready handler. For now, simply implement a function and have it pop up an alert when the button is pressed. Make sure to return `false` so that pressing the `Submit` button doesn't cause an http `GET` request. Save the file and test to make sure pressing the button does indeed trigger an alert dialog box.

Next, implement additional functionality to obtain the values of the `searchString` and `maximumResults` form fields. It might be a good idea to add a test to ensure the user has provided a value in the `searchString` field.

Tip

You can use the `focus()` method to set the cursor focus on an element selected using a JQuery selector

You will want to create a simple JSON object that resembles the `AccountSearchCriteria` class. You will only need to set the `searchString` and `maximumResults` fields for this search. Finally, invoke a JSON Get request, passing the target URL to invoke, the JSON object and the name of the callback function that will be called on successful execution.

Implement another function that will be the callback function (make sure you use the same function name as provided to the AJAX request. Make sure to provide an argument to function to represent the returned results. For now, just have an alert pop up a modal dialog box showing the number of results.

9.2.1.4. Implement a handler method on the `AccountSearchCriteria` controller

In the `AccountSearchController` class, implement a method using the RESTful style. The handler method should include an `AccountSearchCriteria` object in the signature. Use this object to search for matching accounts and return this list as the response body.

Tip

You may want to refer back to the REST sections to see how to do this.

Re-start the server and verify that submitting a search returns results. For example, searching for accounts with the character 'k' should result in an alert dialog showing 3 results were returned. It will be helpful to watch the data exchanged by using Firebug.

Tip

Open the search URL in Firefox. Use F12 to open Firebug. Select the Net tab and ensure that it is enabled. Next, go to the XHR tab. Now, submit the search again and you should see a GET request populated in the display area. Expand the request and examine the Response where you will see the raw JSON object data that was returned. You can also select the JSON tab to get a better idea of how to navigate the returned object.

9.2.1.5. Implement functionality to display accounts

In this final step, you will implement the functionality to display the results, which will be passed into the callback function as a JSON object representing the list of accounts matching the specified search criteria.

In the prior step when monitoring the returned JSON object, you should have observed that what was returned was a list of objects that can be indexed. A logical approach would be to use a `for loop` to iterate over the list and display a row of table data representing the current account values. The heading for the table has already been defined and is epxected to show just the account number and the name of the account owner.

The high level steps you will need to perform then are:

1. Empty out the contents of the former table data
2. Iterate over the results data and display the desired content (you will need to write the HTML for each table row and the table cells)
3. Show the HTML fragment that wraps the entire table

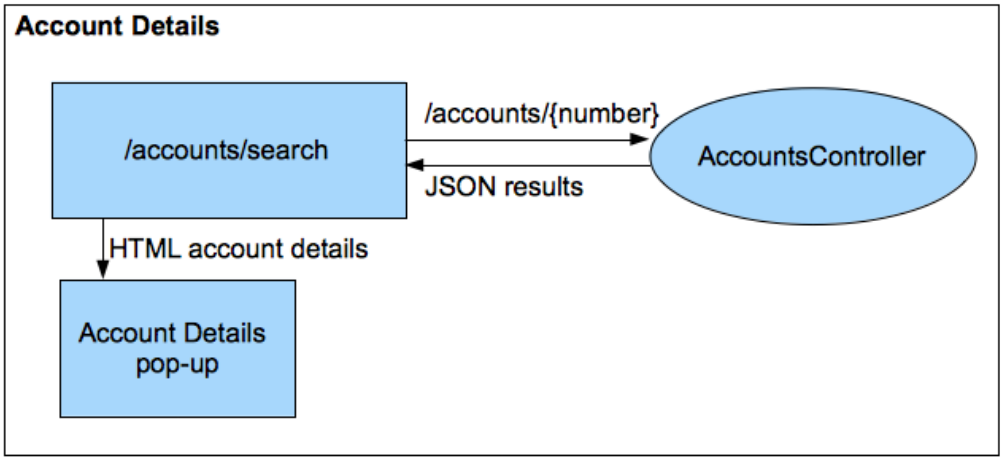
Once you've got this working, you have completed this lab. If you'd like you can proceed with the bonus section.

9.2.2. Bonus: Implement The Account Details Functionality Using AJAX

In this section, you will add additional functionality to display the account details using an AJAX style request

but using the MappingJacksonJson view instead. This will allow you to re-use the existing controller handler functionality that already retrieves the details information.

The following illustration describes the flow of the Account Details scenario:



9.2.2.1. Modify displayed account data to include a link for details

In the prior section, you simply displayed the account number and account name as plain text in a table. In this step, you will modify the display of the account number to include a link that will trigger a javascript event, passing the associated account number.

In your callback function for displaying the table of matching accounts, modify the existing simple table data item for the account number to turn this into a link.

Tip

You will want this to be a link that does not go anywhere but will have an onclick event handler associated to it. Your modified link might look something like this:

`<a href="#" onclick="clickEventHandlingMethod(customValue)">data</a>`. You will want to provide your own name for `clickEventHandlingMethod` and find a way to provide the account number as an argument for each account item.

9.2.2.2. Modify the Spring configuration to support the MappingJacksonJsonView

Before proceeding with your JavaScript coding, you will want to make some changes to your Spring MVC application to support additional view types. You will add a configuration for the Spring `MappingJacksonJsonView` and`ContentNegotiatingViewResolver` classes. Make the JSON view resolver be the default view if one can't be found.

Tip

You may want to refer back to the mvc-view section to see how to do this. You may have also performed a similar task in optional section of the associated mvc-views lab.

Once this is complete, restart the server.

9.2.2.3. Implement a handler function to invoke the AJAX request

To begin, implement the method and use an alert to display the argument that is passed in to the handler. Verify that you are receiving the correct value for each link you click on.

Next, implement the actual handler code. This will use a similar invocation of the JQuery getJSON method call except there is no argument data. Instead, the URL value will be the value of the account number provided as an argument to the 'link clicked' handler function. You can provide the name of a callback function to be called on successful invocation of the AJAX request or you can simply implement the function inline.

Tip

Before implementing the actual code to display account details, you might simply implement an empty callback function and use the Firebug tools to observe the JSON payload that is returned. This should give you some idea as to how to process the data returned. Also, look at the JavaDoc for the `MappingJacksonJsonView` class for additional background.

Now, write some code in your callback function that will extract the result data and display in a simple modal dialog box. Rather than spending a lot of time implementing this in a fancy dialog, simply display an alert with a string containing the text to display where each attribute of the account is displayed with the label and its value on a new line.

Tip

You can use a variable to build up this concatenated string before displaying it as an alert

Once this is complete, reload the JSP and walk through the entire flow. When you are able to successfully display the account details for a selected account, you have completed this step. However, notice that the date value is currently a numeric value. This is because the Jackson/JSON converter is having a little difficulty knowing how to serialize the Date field of the Account object.

9.2.2.4. Help the Jackson/JSON view perform the serialization

To begin, open the `Account` class and look at the various field types. For the most part, these are simple types that the JSON serialization process has no difficulty converting. The one exception is the `dateOfBirth` field, which is a `java.util.Date` type. The default behavior is simply to return the value of invoking the `getTime()` method. In this step, you will be helping the converter serialize this value in a more customized way.

Next, open the `CustomDateConverter` class, which is also in the `rewardsonline.accounts` package. This is a custom serializer class that we will use to help the JSON mapper perform the conversion. This class has one method that performs the serialization. Notice that because this implementation extends a parameterized version of `JsonGenerator` there is a required first argument type of `java.util.Date`. This is a very simple implementation that simply takes the `Date` object and performs the formatting using the `SimpleDateFormat` class and using the pattern 'MM-dd-yyyy'.

In this step, you will simply use this mechanism to tell the JSON mapper how to serialize the date. To do this, you will place the `@JsonSerialize` annotation on the `getDate()` method of the `Account` class.

Tip

You will use the `using` parameter of the `@JsonSerialize` annotation to point to the `CustomDateSerializer` class. Make sure to include the .class suffix on the class name.

Tip

For more information on the Jackson/JSON annotations, refer to the following link:

http://wiki.fasterxml.com/JacksonAnnotations

Once this is complete, reload the JSP and walk through the entire flow. When you are able to successfully display the formatted date for the dateOfBirth field, you have completed this lab and bonus material! Congratulations!

Chapter 10. webflow-getting-started: Getting Started with Spring Web Flow

10.1. Introduction

In this lab, you will set up the Spring Web Flow system in a Spring MVC environment. You will learn how to register flow definitions and start executions of them from your web browser.

What you will learn:

1. How to configure a flow registry and register flows with it
2. How to configure the Web Flow execution engine
3. How to hook Web Flow up in Spring MVC as the request handler for specific resource URLs

Estimated time to complete: 30 minutes

10.2. Instructions

This lab is split into two sections. In the first section, you will become familiar with RewardsOnline use case that is a good candidate for implementing as a web flow. Next, you'll set up the infrastructure needed to execute flows inside the application.

10.2.1. Background

The RewardsOnline application used in this lab consists of a mix of free and controlled navigation. Users may freely search accounts to view and edit. They may also be guided through a controlled process to create new rewards.

Such a controlled "New Reward" process is a good use case for a Web Flow. This process takes place over a series of steps and executes independently of other users. In this section, you will see how the New Reward flow should be invoked by users of the RewardsOnline application.

10.2.1.1. Starting the New Reward Flow from RewardsOnline

A new navigation menu item has been added to the RewardsOnline application that should start the New

Reward flow when selected. You will review this new feature in this step.

First, deploy the `webflow-getting-started` project to the server. Access the the application at http://localhost:8080/webflowgettingstarted. Notice the `New Reward` link in the navigation menu. Select it, and note it fails with a 404. This is because no handler has been registered for that resource URL; web flow, Controller, or otherwise. In fact, the Web Flow system has not been set up at all. In the next section, you will configure Web Flow and then map a new flow definition to the `/rewards/newReward` resource. The flow itself will initially be blank. Do not worry, you will implement its logic later, for now it is just important to understand how flows get registered.

When you are comfortable with how the flow should be invoked from the RewardsOnline application, move on to the next section.

10.2.2. Web Flow System Setup

Now you understand what the RewardsOnline application should be able to do: allow users to create a new reward when they access `/rewards/newReward`. In this section, you'll set up the Web Flow System and learn how to register flows for execution. By the end of this section, you should understand how to configure Spring Web Flow inside Spring MVC.

10.2.2.1. Add one-time system configuration

Web Flow requires some one-time configuration to set it up. You must define two central Web Flow services, and a couple of adapters to plug it into the Spring MVC pipeline. You will do all that in this step.

First, inside `webflow-config.xml` define a `flow-executor` element with id `flowExecutor`. This is the central flow execution engine. It has a number of configuration options, but just use the default configuration for now.

Next, define a `flow-registry` and wire it with your `flow-executor`. This is where you register your flows that can be executed. Each flow is assigned a unique id when registered, and flow ids are mapped to resource URLs. You will see how to register flows later, for now just focus on getting the flow-registry bean defined.

Next, you will register adapters that plug Spring Web Flow into the Spring MVC DispatcherServlet pipeline. These adapters allow Spring MVC to map requests for certain resources to flows.

The first adapter allows a Web Flow to be a Spring MVC request handler, just like a @Controller is another kind of request handler. To install it, define a `FlowHandlerAdapter` bean inside `mvc-config.xml`. The bean will be picked up by the Spring MVC DispatcherServlet automatically.

The second adapter creates resource URL mappings to flows from the contents of your flow-registry. For example, a flow registered with id `rewards/newReward` will be mapped to the resource URL `/rewards/newReward` automatically. To install this adapter, define a `FlowHandlerMapping` bean at the top of

the file.

Tip

Note: be sure to order your FlowHandlerMapping *before* other handler mappings defined in your `mvc-config.xml`. Ordering first allows flow mappings to be queried before mappings to other types of handlers such as Spring MVC @Controllers. Recall mappings are ordered in a chain, and if no mapping is found, the next one in the chain is tried until there is a match or the chain is exhausted.

Tip

When using the `mvc:annotation-driven` tag, handler mappings will be defined starting with order 0. As you cannot change the order of those mappings, make sure you give the FlowHandlerMapping an order of -1.

Now that the core Web Flow infrastructure and Spring MVC adapters have been set up, make sure your application re-deploys successfully. Once you have a successful deployment, return to the welcome page and select the New Reward link again. It will still fail with a 404 resource not found. Even though flow mapping was attempted, no flow was found because you have not defined one yet.

10.2.2.2. Create a New Reward flow from the flow definition template

From here on out, the process of working with Spring Web Flow generally involves authoring your flow definitions, then registering them with the flow registry. Once registered, flows can be executed by accessing their resource URLs.

A web flow is typically authored in a single XML file called a "flow definition". Each flow definition is typically packaged in its own working directory alongside its dependent resources such as JSP templates, message bundles, and other resources. In general, flow definitions should be organized functionally, similar to how you properly structure Java packages. To support automatic refresh of web content without redeployment, flows are typically packaged inside the /WEB-INF directory as well.

With these best practices in mind, consider the specific characteristics of the New Reward flow. This flow is part of the "rewards" functional module, which is distinct from the "accounts" module. A good name for this flow is "newReward", which describes the goal this flow helps the user accomplish. It makes sense, then, that this flow be created in its own directory within the "rewards" namespace. That is exactly what you will do in this step.

Create the New Reward flow definition within a `/WEB-INF/rewards/newReward` directory with the filename `newReward-flow.xml`. Use the following template to define a flow body with a single view state:

```
<?xml version="1.0" encoding="UTF-8"?>
    <flow xmlns="http://www.springframework.org/schema/webflow"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://www.springframework.org/schema/webflow
          http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

    <view-state id="enterDiningInformation"/>

</flow>
```

After creating and saving the flow definition, try accessing the New Rewards link again. Confirm it fails again with a 404. New flows do not automatically get detected; they must be registered in the flow registry. Do that in the next step.

10.2.2.3. Register the New Reward flow

In this step, you will register the New Reward flow, then verify it tries to startup when accessed.

Navigate to your flow-registry bean in webflow-config.xml. There, register your flow located at `/WEB-INF/rewards/newReward/newReward-flow.xml`. Assign the flow the id `rewards/newReward`.

Tip

First, try registering the flow individually using the flow-location element. After you get that working, try using the flow-location-pattern element to register all flow definitions for the application in one concise statement. When you try this, also set the base-path attribute to specify a file-path-pattern relative to a directory such as /WEB-INF.

Once you have registered the flow, select the New Reward link for a third time. You should see a 404 error for `enterDiningInformation.jsp` and a URL with an execution parameter. The flow was launched but the `enterDiningInformation` view failed to render because there is no JSP yet. You have completed this lab. In the next lab, you will implement the real New Reward flow logic!

Chapter 11. webflow-language-essentials: Web Flow Language Essentials Lab

11.1. Introduction

In this lab, you will implement your first web flow to guide users through a process to create a new reward. In-line with Spring Web Flow best practices, you will design and implement your flow's navigation logic first, before adding more complex behavior. You will use mock views provided by your UI designer to quickly iterate on your flow logic with a business analyst. By the end of this lab, you should understand the essentials of the Web Flow definition language, and how to author your own flows.

What you will learn:

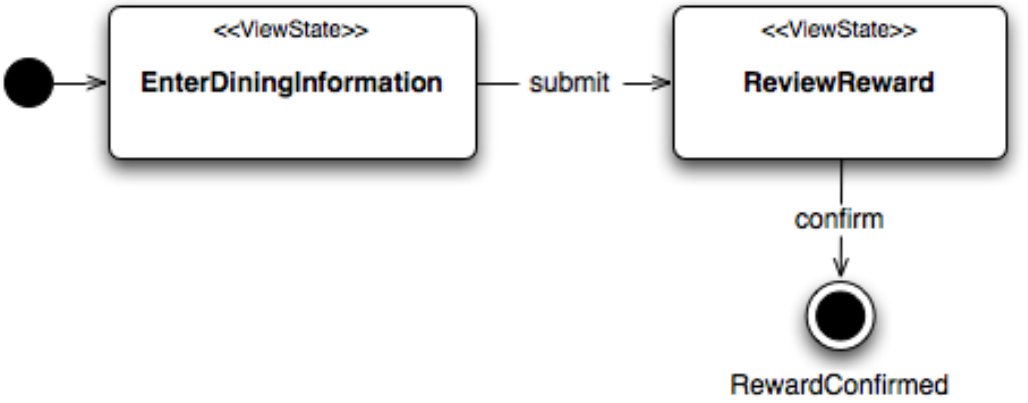
1. How to define the steps of a flow
2. How to trigger transitions that move between steps
3. How to integrate mock views for UI acceptance reviews

Estimated time to complete: 45 minutes

11.2. Instructions

The New Reward flow should guide users through the process of rewarding a member account for dining at a restaurant. The first step of this process should prompt the user to fill out a dining form. On the form, the user must provide the credit card number used to purchase the dining, the restaurant where the dining occurred, the dining amount, and the dining date. After submitting the form, the user should be taken to a screen to review the reward before it is confirmed. After review, the user should be able to confirm the reward. Once confirmed, the user should be redirected to a screen displaying the details of the completed reward transaction.

A graphical illustration of the New Reward flow is shown below:



11.2.1. Render the dining form

In the previous lab, you set up the Spring Web Flow infrastructure and registered a flow definition with an incomplete view state. This lab picks up where the previous lab left off. Confirm this by deploying the webflow-language-essentials project to your server and accessing it at http://localhost:8080/webflowlanguageessentials. Select the New Reward link, and note the 404 error indicating the Web Flow system attempted to render the `enterDiningInformation` view but failed to find the JSP.

Now navigate to the newReward-flow.xml definition in your Package Explorer and open it. As you can see, it has only one view state. Your goal in this lab is to fully implement the flow logic illustrated above in this file.

Get started by completing the first state of the flow that renders the dining form. The UI design team has provided this mock JSP template for you to use during the prototyping phase:

```
<h1>
	Reward an Account for Dining
</h1>

<form id="diningForm" method="post">
	<fieldset>
		<legend>
			Dining Information
		</legend>
		<ul>
			<li>
				<label for="creditCardNumber">
					Credit Card
				</label>
				<div class="control">
					<input id="creditCardNumber" name="creditCardNumber" type="text" />
				</div>
			</li>
```

```
                        <li>
                                <label for="merchantNumber">
                                        Restaurant
                                </label>
                                <div class="control">
                                        <select id="merchantNumber" name="merchantNumber">
                                                <option value="1">Applebees</option>
                                                <option value="2">Subway</option>
                                        </select>
                                </div>
                        </li>
                        <li>
                                <label for="amount">
                                        Dining Amount
                                </label>
                                <div class="control">
                                        <input id="amount" name="amount" type="text" />
                                </div>
                        </li>
                        <li>
                                <label for="date">
                                        Dining Date
                                </label>
                                <div class="control">
                                        <input id="date" name="date" type="text" value="2009-01-20" />
                                </div>
                        </li>
                </ul>

                <button type="submit">
                        Reward
                </button>

        </fieldset>
</form>
```

After you have finished the implementation, open `webflow-config.xml` (Ctrl+Shift+R). Verity that the development mode is enabled. If it is you won't have to restart your server to see the on-the-fly changes you make to your flow definition. When this is done try testing your changes by selecting the New Reward link. You have completed this step once the flow starts and renders the dining form successfully.

Tip

Consider taking advantage of the convention that maps a view-state identifier to a JSP template.

Tip

Spend some time thinking about what makes a good identifier for your state definition. Good state ids are unique to the flow and clearly describe what an actor is expected to accomplish in that step. For example, in this step a user will enter dining information.

11.2.2. Transition to the Review screen

When the dining form is submitted, the user should be taken to a screen allowing him or her to review the reward before confirming it. Complete this step by defining a transition to a state that renders this review screen. The transition should trigger when the submit button is selected on the dining form. The mock screen definition for the review screen has also been provided by the UI design team:

```
<h1>
        Review Reward
</h1>

<form id="reviewReward" method="post">
        <fieldset>
                <legend>
                        Reward
                </legend>
                <ul>
                        <li>
                                <label>Account Number</label> 1234123412341234
                        </li>
                        <li>
                                <label>Reward Amount</label> $10.00
                        </li>
                        <li>
                                <label>
                                        Distributions
                                </label>
                                <table>
                                        <thead>
                                                <tr>
                                                        <td>Beneficiary</td>
                                                        <td>Amount</td>
                                                        <td>Percentage</td>
                                                        <td>Total Savings</td>
                                                </tr>
                                        </thead>
                                        <tbody>
                                                <tr>
                                                        <td>Annabelle</td>
                                                        <td>$5.00</td>
                                                        <td>50%</td>
                                                        <td>$60.34</td>
                                                </tr>
                                                <tr>
                                                        <td>Corgan</td>
                                                        <td>$5.00</td>
                                                        <td>50%</td>
                                                        <td>$34.86</td>
                                                </tr>
                                        </tbody>
                                </table>
                        </li>
                </ul>

                <button type="submit">
                        Confirm
                </button>

        </fieldset>
</form>
```

Tip

Keep in mind the UI design team does not know anything about Spring Web Flow. You may have to make a small change to the screen definitions they provide to encode Web Flow specifics into the form.

When you are done, submit the dining form to test your work. If you got it right the first time, you should be taken to the review reward screen. Once on the review screen, you should also be able to go back using your browser's back button and resubmit. No browser warnings should occur. You have completed this step once you successfully transition to the review reward page, and can go back, forward, and refresh freely.

Tip

Take note of the execution parameter that has been encoded into the flow URL. This parameter identifies the particular *execution* of the New Reward flow definition you are interacting with. Each flow execution is scoped to the user's session and has a unique two-part key in the format e<x>s<y>. The "e" part stands for "execution": the ongoing flow instance you are having a conversation with. The "s" part stands for "snapshot" or "step": a particular step of the flow instance where you can continue from. Notice when you move from the dining form to the review screen, the "s" part changes from 1 to 2, while the "e" part stays the same. This means you are moving from step 1 to step 2 of the 1st execution of the New Reward flow. From step 2, you can go back and resume from step 1.

11.2.3. Confirm the reward

Complete this final step by implementing the reward confirmation navigation logic. From the review reward screen, when the user selects the confirm button the flow should end, then redirect the user to a screen displaying details of the confirmed reward transaction. A @Controller to handle showing confirmed rewards has already been written for you, so you just need to have the flow redirect to it. After the flow has completed, it should not be possible to go back and resubmit the same reward transaction.

Tip

Spend some time thinking about what makes a good end-state identifier. End-states describe flow outcomes, or results. They communicate an overall outcome that has already happened, something you potentially want to report back to the flow's caller. In this case, the outcome is the reward has been confirmed.

Tip

The special externalRedirect: directive can be used in conjunction with the view attribute to request a redirect to another resource from an end-state.

Tip

Send a mock reward confirmation number through the redirect for now.

Tip

After a flow execution ends, its execution key is invalidated and allocated resources are cleaned up. You can confirm this by going back in your browser--all previous snapshots of execution 1, for example, are no longer resumable; they have been cleaned up automatically by the system. When you click the New Reward link again, notice how the execution number now increments from 1 to 2, etc, indicating you are starting entirely new flow executions.

After confirming the reward, once you are successfully redirected to the confirmed reward screen and cannot go back, you have fully implemented the flow logic and completed this lab!

Chapter 12. webflow-actions-1: Web Flow Actions 1 Lab

12.1. Introduction

In this lab you will be introduced to actions inside Web Flow. In the previous lab, you created a basic flow that outlined the steps to complete the task. Now you will apply behavior to the flow, processing the user input and integrating with the `rewardNetwork` back-end infrastructure.

What you will learn:

1. How to define variables that persist during the flow execution
2. How to apply automatic binding to a model object for a view-state
3. Evaluate actions in different phases of the flow

Estimated time to complete: 60 minutes

12.2. Instructions

Currently the flow consists of static views connected with transitions. The transitions are initiated with button clicks that in turn are translated into Web Flow events. Although the views are static, the navigation is driven with an actual flow instance, which means you have developed and road-tested your navigation logic. With that out of the way you can turn your attention to flow behavior.

12.2.1. Making the flow dynamic

Initially you will declare a flow variable that will be used to collect dining information. You will bind this variable to the `enterDiningInformation` view and apply type conversion and validation as necessary. Then you will invoke the `diningFormDataProvider` repository to load and expose a list of `Restaurant` objects to fill the Restaurants drop-down. Lastly you will invoke the `rewardNetwork` one time to calculate reward contributions before the review page and a second time to create the reward once the user confirms the change.

12.2.2. Collect Dining Information

Begin by deploying the `webflowactions` project and then proceed with the steps below.

12.2.2.1. Declare a flow variable

You first task is to declare a `DiningForm` variable. Open `newReward-flow.xml` and add the variable giving it the name `diningForm`. Variables declared here will remain available for the duration of the flow.

12.2.2.2. Add Spring form tags

Open `enterDiningInformation.jsp`. Make the form dynamic using the Spring form tag library and use the new flow variable as the model attribute for the form. You can begin by adding the necessary taglib declaration at the top of the page. Then go through and convert all form tags from plain HTML to Spring form custom tags. And don't forget to enable showing field specific errors.

Tip

To enter a taglib declaration, copy the previous taglib declaration, place your cursor at the start of the uri value and use Ctrl+Space to pick a new uri.

Tip

For the time being you can create the restaurant <form:select> using a path attribute only. In the next step you will make sure it's populated.

When this is done re-enter the flow using the New Reward link. This will start a new flow and cause the `diningForm` variable to be created. If your changes are correct you should see a blank form with empty fields. The form is empty because the newly created `diningForm` variable does not contain any values.

12.2.2.3. Populate the Restaurants drop-down

Before the drop-down can be populated you must retrieve the data required to populate it. Open `JdbcDiningFormDataProvider` and review its content. Observe the name of this Spring bean and the method that finds all restaurants. This method returns a map of restaurant names keyed by merchant number. That is all you need for a simple drop-down.

Next open `newReward-flow.xml` and add an action that is invoked when the `enterDiningInformation` view is rendered. The action must use the above method to find all restaurants and save them to a variable so it is accessible to the view.

Tip

When saving to a variable remember that Web Flow has several reserved EL variables. Those are `flowScope`, `viewScope`, `flashScope`, `requestScope`. Try to select the most appropriate one. If you are not sure ask the instructor.

When this is done initialize the `items` attribute of the restaurants <form:select> from the saved variable. Refresh the page and verify the drop-down has live data.

12.2.2.4. Submit the form

Enter some data and submit the form.

Tip

Don't forget that credit card number 1234123412341234 is a valid credit card in the database.

Return with the browser back button and observe the form is blank. If the submitted values were applied correctly through data binding the form wouldn't have been blank. To enable data binding add a `model` attribute to the `enterDiningInformation` view state and point it to the `diningForm` variable.

Submit the form again. This time if you return you should see a form with the values you entered. This confirms data binding is taking place.

12.2.2.5. Check type conversion

You may have already noticed you cannot submit invalid amounts or dates (if not give it a try.) How does this work? Open `TypeConversionService.java` and look at the `installFormatters` method, which adds two Formatters. The formatters are in the same class so you can examine their content as well. Now open `webflow-config.xml` and see how the conversion service is registered with the `flowBuilderServices`. Open `mvc-config.xml` and look at the `mvc:annotation-driven` definition to see how the same conversion service can be used for Spring MVC conversion.

Tip

If you're looking for the bean definition for the TypeConversionService itself, remember that the class is annotated with @Component, triggering automatic bean registration. Annotation-driven beans can be referenced in XML bean definitions (such as the `conversionService` bean.

If you want to experiment remove the conversion service from your `flowBuilderServices` and verify the form fails due to type conversion errors. When you're satisfied put the service back in and proceed to the next section.

12.2.2.6. Add required fields

Try submitting without any values to see what happens. All fields are indeed required but the flow will take you to the review page with or without values. Fortunately the flow definition syntax allows adding required fields for a given view state. Go ahead and use this syntax to designate all required fields of the `diningForm` variable.

When that's done try submitting again. This time you should remain on the same page and see required field errors.

Tip

If you remain in place but don't see errors you probably didn't add any <form:errors> tags in the previous step.

12.2.2.7. Add custom validation

Recall that Web Flow will automatically invoke validation on a model. One convenient place to add you validation logic is directly in the model object. Refer to the presentation slides on the exact signature and add a validation method to `DiningForm.java`. This method should verify that a credit card number is 16 characters long. If not it should register a field-specific error using the error code `error.invalidFormat.DiningForm.creditCardNumber`. Verify the change by entering an invalid credit card number.

12.2.3. Review Reward

In this section you will make the Review Reward page dynamic.

12.2.3.1. Invoke the RewardNetwork to calculate account contributions

Open `newReward-flow.xml`. Add an action to call the Reward Network `calculateContributionFor` method and store the result as a variable. This action must be invoked at some point before the `reviewReward` view state is rendered.

Tip

Recall that you can embed actions in various places in a flow - during a transition, upon entering a state, before rendering a view state. Try to select the most appropriate place for adding this action.

Tip

You'll probably need the `DiningForm createDining()` method to create the Dining object required as input to the Reward Network.

Tip

Once again consider the best scope to store the resulting account contribution in.

Verify the change by re-submitting the form and check for exceptions. When you can re-submit without exceptions you are ready to move on.

12.2.3.2. Display the account contribution

Open `reviewReward.jsp` and make it display actual account contribution data. Remember this page is read-only. There is no need to use the Spring form tags. Just use EL expressions like ${accountContribution.amount}. When you're done or at any point along the way use the browser to verify the changes.

12.2.4. Create the reward

After the user presses Confirm, you need to make sure the reward is created, and a redirect is made to an external confirmation page with the actual confirmation number.

12.2.4.1. Invoke the rewardNetwork to create the reward

Open `newReward-flow.xml` and add an action to create the reward after the user has confirmed the changes. Check the methods of the Reward Network for the appropriate method signature. Save the return value as a variable that will be accessible in the end state where you'll need the confirmation number.

12.2.4.2. Redirect with the confirmation number

It's common for a flow to perform a redirect to an external resource. Recall that the end-state currently uses a hard-coded confirmation number. Make this number dynamic by using the variable you created in the previous

step. To do that you'll need to embed an EL expression within the view string. For example:

```
view="externalRedirect:/resource/#{foo.number}"
```

Confirm the changes in the browser.

Tip

You will need to modify RewardsController so that the method responsible for handling this request actually returns the Reward object for the given confirmation number. There is already a protected method available that you can call to accomplish this.

Tip

You will also need to modify show.jsp so that it displays the actual values from the reward instance rather than the mock values that are currently being displayed.

Congratulations, you've now completed this lab.

12.2.5. Bonus Work

You have now completed the lab but have the choice to continue with the following bonus material.

12.2.5.1. Complete the flow unit tests

Open `NewRewardFlowTests` and review all the TODO comments. On a high level there are two things to do. Register the Spring beans referenced in the flow definition and test for the presence of variables created by actions. Fortunately stub implementations for both Spring beans are already available as inner classes. Have a look at them and then work on adding the tests and making them green.

Chapter 13. webflow-actions-2: Web Flow Actions 2 Lab

13.1. Introduction

In the previous lab you learned how to make flows dynamic with the help of actions. In this lab you will work on more advanced scenarios with actions including exception handling.

What you will learn:

1. Deal with exceptions using a MultiAction
2. Reuse a sequence of actions with an action state
3. Add decisions to flow's algorithm

Estimated time to complete: 60 minutes

13.2. Instructions

This lab is divided into several sections. In the first section you'll review the goals for this lab. Subsequent sections will guide you through the details of the implementation.

13.2.1. Existing Code and New Requirements

Begin by deploying the `com.springsource.training.webflowactions2` project to the SpringSource tc Server.

13.2.1.1. Flow definition

Open `newReward-flow.xml`. As you recall the flow declares a `DiningForm` variable and binds it to the "Enter Dining Information" view. This enables data binding and validation. In addition the flow also invokes several actions. It loads restaurants when the initial view is rendered. It calculates contributions before getting to the review page. Lastly it creates the reward upon confirmation.

13.2.1.2. Invalid credit card exceptions

One of the things to consider when invoking actions is how to deal with exceptions. To illustrate this go to

your browser, create a new reward, and enter an invalid credit card number (e.g. 1234123412341233). Press the Reward button and examine the resulting exception. It is `ActionExecutionException` and it's thrown while evaluating the expression `rewardNetwork.calculateContributionFor`. Scroll further down and see that the root cause is an `InvalidCreditCardException`. The flow currently does not attempt to deal with this expected exception.

After some consideration it has been decided how to address the issue. The application must catch the exception, record an error message, return to the Enter Dining Information page and show the error next to the Credit Card Number field.

13.2.1.3. New application requirements

After reviewing the results from the last iteration, the business users have formulated two new enhancement requests. The first is to add a checkbox on the "Enter Dining Information" page. The idea is to allow advanced users to skip the "Review Reward" stage and proceed directly to create the reward.

The second request is to enhance the Review Reward screen to make it easy for users to correct dining information They should see recalculated amounts without going back and without reloading the current page.

Now that you have your goals set, the remaining sections will guide you through the implementation.

13.2.2. Handle the InvalidCreditCardException

You're already familiar with the issue. The approach you'll use will be to wrap the RewardNetwork call with a thin layer of Java code. The goal is to stop the exception, record a field-specific error and remain in the same view.

13.2.2.1. Introduce a MultiAction

Web Flow provides a base class called `MultiAction`, which can be used for web layer logic. A `MultiAction` has one or more methods with a signature similar to the one below:

```
public Event someMethod(RequestContext context) {
    try {
        ...
        return success();
    } catch (SomeException e) {
        ...
        return error();
    }
}
```

Go ahead and create a class with a method that will wrap the invocation to the RewardNetwork to calculate the contributions. Call the class `NewRewardActions` and add it to the package `rewardsonline.rewards.newreward`. The method needs to obtain the diningForm, invoke the RewardNetwork, and store the result as a flow scoped

variable. It should also catch `InvalidCreditCardException` and add a field specific error. Below is an example of recording a field-specific error:

```
context.getMessageContext().addMessage(
    new MessageBuilder().error().source("fieldName").defaultText("Text to display").build());
```

Remember to configure the new class as a Spring bean and also to dependency-inject it with a RewardNetwork instance.

13.2.2.2. Confirm the bug is fixed

When the method is ready, go back to `newReward-flow.xml` and replace the call to `RewardNetwork.calculateContributionFor()`. Rather than using an `action-state`, simply call the method directly for now. We'll add action states later. Lastly use the browser to verify the change. Using an invalid credit card number should keep you on the same page and show the error message.

13.2.2.3. Add method to wrap the rewardAccountFor() method

Now that you've done this once, use the same approach to wrap the other service-layer invocation. That's the call to the `RewardNetwork.rewardAccountFor()` in the final transition. This method invocation is exposed to the same issue.

13.2.3. One-Click Reward

Currently the flow assumes the reward will be created in two steps. For example when dining information is submitted, the transition calculates the contribution amounts. To support the new requirement you will need to add a checkbox to the view and at the same time introduce a decision point in the flow definition.

13.2.3.1. Add a checkbox to the view

Open `DiningForm.java` and verify it already contains the field `oneClickReward`. Next open `enterDiningInformation.jsp` and create a checkbox using the Spring form tags and bind it to the `oneClickReward` field.

13.2.3.2. Add a decision state and action states

Open `newReward-flow.xml`. Your goal is to introduce a decision state between the `enterDiningInformation` and the `reviewReward` states. The decision state should test the `diningForm.oneClickReward` field and result in one of two outcomes. Either reward the account for the dining event and proceed to the end state. Or calculate the account contribution amounts and proceed to the `reviewReward` state.

Tip

You will need to define action states to model those outcomes. Actions states can contain one or more actions and also deal with standard success or error events as follows:

```
<action-state id="...">
    ...
    <transition on="success" to="abc" />
    <transition on="error" to="xyz" />
</action-state>
```

When you're done making the change test it in the browser.

Tip

If your flow still goes to the review reward page even when the checkbox is selected, make sure you added a binder statement to the enterDiningInformation state for the new field.

13.2.4. Re-Calculate the reward on the Review Reward page

To fulfill this requirement you need to add the dining form to the review page. Open `reviewReward.jsp`. Right now it contains a single, read-only form with contribution amounts. It needs to have a second form with the dining information.

13.2.4.1. Add the dining form to the reviewReward.jsp

Use <tiles:insertTemplate> to insert `enterDiningInformation.jsp` at the bottom of `reviewReward.jsp`. After that go back to the browser and navigate to the review page. The dining form should display but the date will not be decorated correctly and the restaurant drop-down will be empty. There is a different reason for each of these problems. The date fails because the data binding is not set up in the reviewReward view state. The restaurant drop-down fails because the list of restaurants has not been loaded. Open `newReward-flow.xml`, correct both issues, and verify the fix the browser.

Now that the form is displaying correctly focus on the Reward button. The text "Reward" makes sense on the first page but here it should probably be "Recalculate". Open `enterDiningInformation.jsp` and find the button. Notice it has been parameterized so you can influence the text of the button and the event it generates through request attributes. Go back to `reviewReward.jsp` and add the following before the dining form is inserted:

```
<c:set var="rewardButtonEvent" value="recalculateReward" scope="request" />
<fmt:message key="command.recalculateReward" var="rewardButtonText" scope="request" />
```

Refresh the page in the browser. The button is now called "Recalculate Reward" and the event it submits is

"recalculateReward".

Note that when you view the page, the One-Click reward button is present. While this is important for the Enter Dining Information page, it is fairly meaningless in the review reward page. Modify `editDiningInformation.jsp` to not display the button in this case (you should be able to use the value or lack thereof of `rewardButtonEvent` to determine this)

13.2.4.2. Re-calculate the contribution amounts

Now that the form is ready change the amount and submit. You will the see the error message:

```
No transition found on occurrence of event 'recalculateReward' in state 'reviewReward'.
```

Analyze the message and fix the flow definition accordingly. When you have the right fix the page will correctly display the re-calculated contribution amounts.

13.2.4.3. Simplify the Review Reward view

Your new Review Reward page now has two distinct headers - "Review Reward" and "Reward Account for Dining". This is a bit messy, and ideally we'd like to remove the second header. To do so, create a new file (call it `diningForm.jsp`) and move the dining form from `editDiningInformation.jsp` into it. Now use Use <tiles:insertTemplate> to insert `diningForm.jsp` into `editDiningInformation.jsp`. Finally, just insert `diningForm.jsp` (instead of `editDiningInformation.jsp`) into `reviewReward.jsp`. When this is done verify both pages still load correctly.

You've now completed this lab.

Chapter 14. web-security: Secure Your Web Application With Spring Security

14.1. Introduction

In this lab you ensure users have signed in before they can access the application. After signing in, regular users are allowed to browse but not edit any data. Users with administrative privileges on the other hand can edit existing accounts and create new rewards.

What you will learn:

1. How to deploy Spring Security with a servlet filter
2. How to configure Spring Security to protect specific URL patterns
3. How to configure a form-based login
4. How to use the Spring Security custom tag library for conditional rendering

Estimated time to complete: 45 minutes

14.2. Instructions

The instructions in this lab are divided into 2 parts. In the first part you will deploy and configure Spring Security. In the second part you will protect account editing and new reward creation with administrative privileges.

14.2.1. Deploy and Configure Spring Security

Spring Security protects web applications using a chain of standard servlet filters. The Spring Security filter chain is initialized in a Spring application context. Creating the filters in a context is better than declaring them in web.xml because Spring provides much more flexibility in how filters are to be initialized, what dependencies they should have, and so on. Spring offers a class called `DelegatingFilterProxy`, which can help with that.

14.2.1.1. Declare the DelegatingFilterProxy in web.xml

Open `web.xml` (Ctrl+Shift+R) and add a new filter of type `org.springframework.web.filter.DelegatingFilterProxy` with the filter name `springSecurityFilterChain`. Map this filter to all incoming requests using the /* pattern.

Note

The DelegatingFilterProxy works by delegating to a Spring-managed bean that implements the Filter interface. The filter name should match to the bean name in the Spring application context. All calls to the filter proxy will then be delegated to that bean in the Spring context.

Now that we've deployed the filter let's see what it'll do. Click and drag the project to the server. If you try to accessing any application pages you'll see a 404 error. That's because the filter is deployed but fails to find a bean named `springSecurityFilterChain` in the Spring configuration.

14.2.1.2. Configure The Spring Security Filter Chain

Enabling the Spring Security filter chain is quite easy and we can do so from a Spring application context using Spring Security's custom XML namespace.

In `web.xml` notice the `contextConfigLocation` context parameter near the top specifying Spring configuration files to be loaded into the root Application Context at startup. This should list `/WEB-INF/spring/security-config.xml` only. Navigate to that file and open it. The file should be mostly empty except and not have any security configuration yet.

Insert a new line somewhere between the opening and the closing <beans:beans> root element and press Ctrl+Space. STS has knowedge of the Spring Security namespace and will offer typing suggestions. Begin typing or scroll down to the <http> element and select it. The <http> element has an attribute called `auto-config`. Use Ctrl+Space to add this attribute and set its value to 'true'. Also, turn on support for Spring Security SpEL expressions by adding the attribute `use-expressions="true"`

Tip

As the name suggests, setting auto-config to true configures a filter chain with a default set of services. Move your mouse over the auto-config attribute to get some more information on what these default services are or refer back to the presentation.

After you save your changes restart the server. Once server startup is complete, you will still get a 404, but there will be a new error in the console. This time you will get the following exception:

```
No bean named 'org.springframework.security.authenticationManager' is defined
```

This exception is coming when Spring tries to set up Spring Security. It's telling us we need to provide an Authentication Manager (the source of user information). Fortunately this is also easy to do.

14.2.1.3. Configure an In-Memory User Details Service

For this lab we will use an in-memory user details service. However, we could have also used one that goes to a database or an LDAP repository instead.

In `security-config.xml` use Ctrl+Space to add <authentication-manager>, <authentication-provider>, and <user-service> elements all nested within each other. Within <user-service> insert a <user> element setting its user name to 'joe', the password to 'spring', and the authorities to 'ROLE_USER' . You now have a declared a simple in-memory user service with a single user.

After the server has restarted, verify that application pages are displayed correctly. They are protected with Spring Security's chain of filters, but we have not yet defined any specific URL patterns that need to be protected.

14.2.1.4. Specify URL Patterns To Protect

In `security-config.xml` find the <http auto-config="true" use-expressions="true"> element and insert a <intercept-url> element inside of it. Set the pattern attribute of this element to "/**" and the access attribute to `hasRole('ROLE_USER')`. This indicates that every URL of the application now requires the specified role.

Restart the server and try to access the application pages. You should see the following:

Figure 1: Default Spring Security Login Page

This is the default Login page generated by Spring Security. Use it to login and you should be able to succeed. This works but what you'll probably want to do is provide Spring Security with your own login page that looks a little more like the rest of your application.

14.2.1.5. Configure Form-Based Login

Open `login.jsp` and review its content. This page is functionally identical to the one Spring Security generated automatically but it has styles added to it.

In `security-config.xml` insert another line with the <form-login> element (right below the <intercept-url>) and set its `login-page` attribute to `/login`. Restart the server.

When Spring Security logged you in previously, it added a cookie to your session with your login information. Even if you restart the server, this will retain your credentials, preventing the login page from being displayed. Make sure you clear this cookie out of your session by selecting "Clear Session Cookies" from the "Cookies" menu of the Web Developer Toolbar.

Now try to access the default page of the application. If you're using Firefox you will likely see the following:

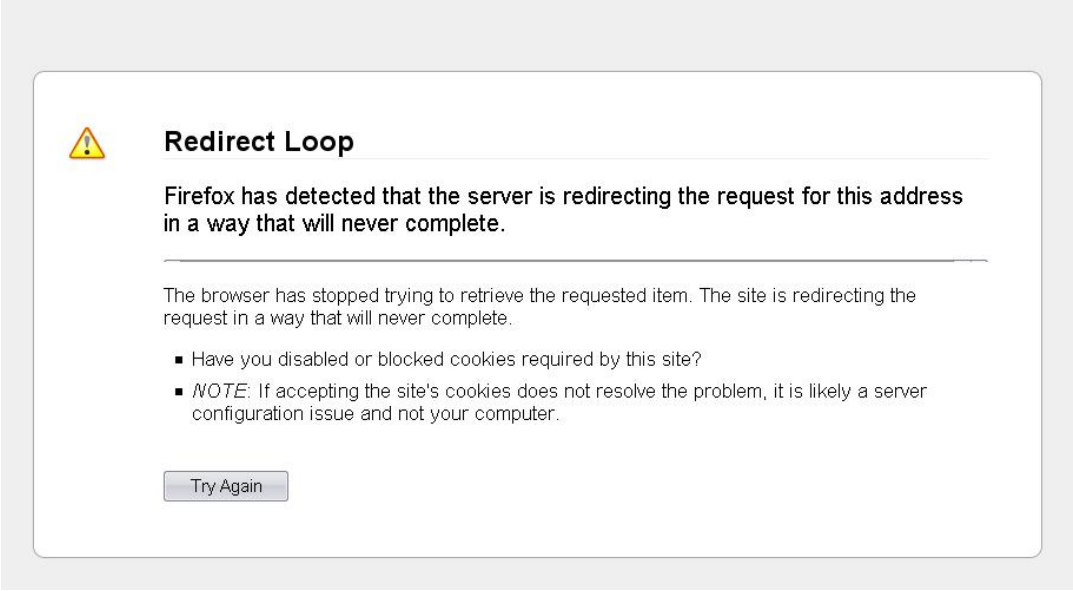


Figure 1: Redirect Loop Message In Firefox

This one is a bit harder to explain, although if you look through your security configuration and think it over you might be able to guess what's going on. Or if you'd rather read about it, then just keep on.

The problem is that we've protected all pages that match "/**" - that includes the login page too. Now that we've provided Spring Security with a login page, each time Spring Security tries to go there it fails because it's protected, and redirects to the login page so the user can authenticate. The result is a loop that doesn't end.

To fix this add another URL pattern that looks like this:

```
<intercept-url pattern="/login*" access="isAnonymous()" />
```

`isAnonymous()` is a Spring Security SpEL expression identifying anonymous users. If you recall one of the default services in the Spring Security filter chain assigns an anonymous identity to incoming requests if they are not yet authenticated.

Tip

If you still get the "Redirect Loop" message, remember that intercept-urls are matched *in the order listed*. If /** is first, it will match all URLs, regardless of whether other matches exist.

Try one more time. This time the login page should appear although it won't be appear to be properly styled. Perhaps you've guessed the problem. You need to allow access to static resources. Modify the security configuration and add the following additional <http> elements:

```
<http pattern="/images/**" security="none" />
<http pattern="/styles/**" security="none" />
<http pattern="/resources/**" security="none" />
```

Try out your changes. You should see a properly styled login page that allows you to enter and use the application.

14.2.1.5.1. Logging Out and Dealing With Login Errors

How do you log out after you've logged in? Try this URL:

```
http://localhost:8080/websecurity/j_spring_security_logout" />
```

What we'd like to do is make that easier for the user. Open `standard.jsp`, find the div with id "nav". Within it is an unordered list of navigation items. The last item with the TODO is where you can add the Logout link. The link could look like this:

```
<a href="<c:url value="/j_spring_security_logout"/>">Logout</a>
```

Go ahead and test it out. Also notice the use of the Spring Security tag called authorize, which renders the logout link only if the current user does not have the anonymous role.

For one last bit before moving to the next section, what happens if you fail to login? Clear your cookies again, try it out, and see for yourself. We'd like to render an error when the login fails. Check the login page to see how the login error is rendered. If the page is triggered with the parameter `login_error` a message will be rendered. We can force this error by adding a "authentication-failure-url" attribute to the login-form tag. Set that attribute to the value "/login?login_error=1

14.2.2. Protect Parts of the Application With Administrative Privileges

14.2.2.1. Add User With Admin Role

In `security-config.xml` add a new user called "admin" and password "springsource" having the authority "ROLE_ADMIN". Note that while we could have named the authority anything we like, the only requirement is that it starts with "ROLE_" so that Spring Security understands this authority is a role it can vote on.

Next add two more <intercept-url> elements to protect `/rewards/newReward` and HTTP PUT requests for `/accounts/*` with "hasRole('ROLE_ADMIN')". The former is the URL of the flow for creating a new reward. The latter is the URL for submitting account changes. We want both of these to be limited to admin users.

Tip

Remember that the order of the intercept-urls is important. More specific URL patterns precede more general ones.

Try to login as the user "joe". You should be able to browse the application, but will get a 403 (Access Denied) error if you try to create a reward or save account changes. That's because those operations require ROLE_ADMIN. Now try logging in as "admin". This time you'll see a 403 error when accessing the home page. That's because /** is protected with ROLE_USER. Tweak the intercept URL patterns. You're done when users with "ROLE_ADMIN" can browse and edit everything. Users with "ROLE_USER" can browse but not edit anything.

14.2.2.2. Use Spring Security Tags For Conditional JSP Page Rendering

You've probably noticed that even as a regular user you can still get to the Edit Account page. There is no harm in that but it would be better if the Edit link was indeed hidden. Similarly, there is no point in showing the New Reward link if we don't have the rights to use it.

Open `accounts/show.jsp`. Find the Edit link and surround it with a <security:authorize> element. Set its access attribute to "hasRole('ROLE_ADMIN')". Now this link will only be rendered for admin users. Go to the Edit Account page as a regular user (or refresh it if already there). This time you should not see the link.

Open `standard.jsp`. Find the link for creating new rewards and repeat the same steps as above. Refresh and verify the link is not shown for regular users.

You've now completed this lab.

Chapter 15. roo: Getting Started with Spring Roo

15.1. Introduction

This lab will get you started with Spring Roo. In it, you will implement a simple domain model with a scaffolded web interface and some other functionality to see how easy it is to get up and running with Roo.

What you will learn:

1. How a Spring Roo application is set up
2. How to set up JPA and add some entities
3. How to set up Spring MVC and add scaffolded controllers and views

Estimated time to complete: 60 minutes

15.2. Instructions

The instructions in this lab are divided into sections. In the first section you'll set up a new Spring Roo project and JPA persistency. In the next section you will add two entities with some constrained fields and a one-to-many association. In the third section you'll scaffold a web interface to perform CRUD operations on these entities. A final optional bonus section shows some ways to customize the generated interface.

15.2.1. Setting up a new Spring Roo project

When you open the 'roo' working set, you'll notice that there is only a solution project. That's because you will create the start project yourself.

15.2.1.1. Step 1: Create a new project

Right-click on the 'roo' working set and choose "New -> Spring Roo Project". In the wizard that pops up, use 'rewardsonline' as both the Project name and the Top level package name. Check the "Add project to working sets" checkbox and choose the 'roo' Working set. Then click "Next" and "Finish" to create your project.

Notice that a Roo Shell View is automatically opened for your new project. When you want to enter a Roo command you can do it in this shell.

Note

Typing 'Ctrl+R' should open up a pop-up prompt that allows you to enter Roo commands as well, but this stopped working for the author of this lab in STS 2.7.1 on Windows.

15.2.1.2. Step 2: Set up persistency

All Spring Roo 1.1.x projects use JPA 2 for persistency, but you'll have to tell Roo what provider and database you're using. Type 'hint' in the Roo shell to see instructions on how to do that, and then follow these instructions using "HIBERNATE" as the provider and "HYPERSONIC_IN_MEMORY" as the database. Remember to use Ctrl+Space (Cmd+Space on a Mac) to get code assist for completing commands and command arguments!

Note

The stand-alone Roo shell uses Tab for completion, but in STS the default Eclipse key bindings are used instead.

Note the files that are added and updated to add JPA support to your project: this saves you a lot of tedious work in setting things up yourself.

15.2.2. Adding some entities

You now have an empty project that uses Spring and JPA: the next step is to add some entities that model the domain of the application.

15.2.2.1. Step 3: Add an Account

First type 'hint' again: Roo suggests that you add an entity using the 'entity' command. Create an `Account` entity under the `rewardsonline.accounts` package: you can do that by specifying the class as `~accounts.Account` (the '.' between the tilde and the package name is optional). Also, follow the hint instructions and add an optional argument called 'testAutomatically'. To see optional arguments, first type '--' (two dashes) and then use code assist.

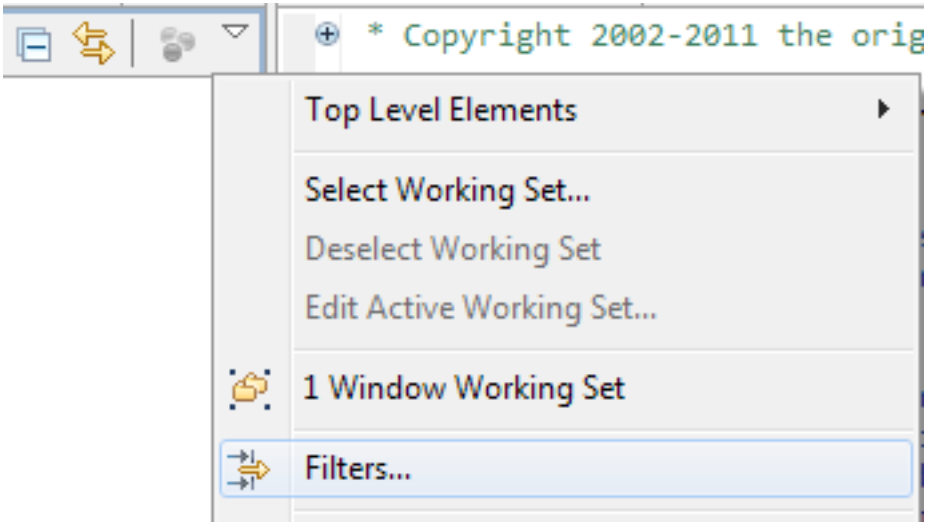
Note the files that Roo creates for you: in addition to the `Account.java` there are several .aj files and there's an integration test with a so-called data on demand class created for you. The latter allows for easy creation of test data.

15.2.2.2. Step 4: Add some fields

Add two private `String` fields to the `Account` class, `name` and `number`. You can either do this on the Roo command line using the 'field string' commands, or directly in the Java editor:

- When using the Roo shell, add a '--notNull' to the `name` field and a '--unique' to the `number`. To use 'number' as a field name, you'll also need to add the '--permitReservedWords' argument.
- When using the Java editor directly instead, annotate the `name` field with `@NotNull` and the `number` field with `@Column(unique=true)` (make sure to import the `javax.persistence.Column` type and not the one from `org.hibernate.annotations`!).

When you're done, the `Account` class still looks fairly empty: just some fields and annotations. That is because Roo places the supporting code it generates for the `@RooXxx` annotations in hidden AspectJ files containing inter-type declarations (ITDs). To see these files, click the small downwards-pointing arrow in the Package Explorer View and choose "Filters...". Then deselect the "Hide generated Spring Roo ITDs" element and press OK.



Have a look at the generated code: it reads almost like Java, but this code adds fields and methods to the `Account` class during compilation. These files are internal to Roo, so you should never edit them: instead, you can simply provide your own implementation of some method or field and Roo will automatically remove the corresponding ITD. You can try this by implementing your own `toString` method in `Account`: as soon as you save that change, Roo will remove the corresponding .aj file. This file will be restored when you remove your method again, since the class still has the `@RooToString` annotation. When you remove a @RooXxx annotation, the corresponding code will no longer be generated for you.

15.2.2.3. Step 5: Add a Beneficiary with some fields

Now create another entity in the same package called `Beneficiary`. Add a non-null String field `name` and a `BigDecimal` field `savings` with a decimal minimum of 0. When you use the Roo command line for this, that requires the 'field number' command and the '--decimalMin' argument; when using the Java editor, use the `@DecimalMin` JSR-303 annotation.

By default Roo simply appends an 's' to an entity's name to come up with a plural form: for "Beneficiary" that's wrong. Annotate the class with `@RooPlural` and specify "Beneficiaries" as its value attribute. You must do this in the Java editor.

15.2.2.4. Step 6: Add a bi-directional association between Account and Beneficiary

An account has 1 or more beneficiaries. To be able to work with such a relationship in a Roo-generated web interface, you have to configure a bi-directional association: that means that a `Beneficiary` must have an `account` field and an `Account` must have a `beneficiaries` field. Add these fields either using the Roo command line or in the Java editor:

- In the Roo shell, use the 'field ref' command to add an `account` field of type `Account` to the `Beneficiary` class. Then use the 'focus' command to switch to Account and use the 'field set' command to add a `beneficiaries` field: specify a cardinality of ONE_TO_MANY!
- Or use the Java editor to add the fields: annotate the `account` field with `@ManyToOne` and the `beneficiaries` field (which should be of type `Set<Beneficiary>`) with `@OneToMany(cascade=CascadeType.ALL)`.

With this setup you have finished a simple domain model that you can now generate a Web UI for.

15.2.2.5. Scaffolding a Spring MVC web interface

So far, your project is a simple Java project that would produce a jar file when built. In this section you'll turn it into a web project with a Spring-MVC backed frontend that's created using Tiles and JSPs with custom tags: all technology that you learned about in the course so far.

15.2.2.5.1. Step 7: Set up Spring MVC support

Enter the 'web mvc setup' command to add support for Spring MVC to your application. Notice the long list of files that's added and updated: this simple command has turned your project into a web project (including Maven support for deploying to a local Tomcat or Jetty instance using 'mvn tomcat:run' or 'mvn jetty:run'), added Spring-MVC to the dependencies, configured a `DispatcherServlet`, added Tiles 2 configuration and a default layout including static resources, and added custom tags, a resource bundle for I18N purposes and some default error pages.

Before you continue, browse through the configuration and generated files to see al the things that Roo has set up for you and to get a feel for how the application works. Most of what you see should be familiar from the earlier modules in the course.

15.2.2.5.2. Step 8: Scaffold views and controllers

To get up and running quickly, Roo supports the so-called scaffolding of views and controllers for your web application. This means that basic CRUD (Create-Read-Update-Delete) support for your entities will be provided by a generated user interface. Roo will continue to track changes to your entities and will update the views and controllers accordingly.

Use the 'web mvc all' command to scaffold a web UI with '~web' as the package to use for the generated classes.

You can now deploy the application to the local Tomcat instance and run it. Navigate to http://localhost:8080/rewardsonline to see what the scaffolded UI looks like. Create an Account and then create some Beneficiaries that you add to that Account. Notice how an Account is shown in the list when creating a new Beneficiary: you see a combination of its properties. Notice the same thing happens for beneficiaries when you look at an Account.

Also, notice how the 'notNull' constraints you specified earlier are reflected in the user interface: for not-null fields an extra message is shown, and client-side validation is active which also prevents you from submitting an invalid form.

Explore the user interface a bit more and then proceed with the next step.

15.2.2.6. Bonus Section: Modifying the scaffolded interface

The user interface that has been generated is not typically intended to be used as-is: it's just a quick way to get started, but you are still responsible for designing and implementing your web interface of course. One thing that's somewhat annoying is that accounts and beneficiaries are displyed using a combination of their attributes at the moment: it would make more sense to just show their names instead.

15.2.2.6.1. Step 9: Define your own Converters

Spring Roo uses Spring Converter implementations for rendering entities as labels. These converters are registered in a class called `ApplicationConversionServiceFactoryBean`: please open that class now. Notice that there are no converter definitions here: like with entities, this code is woven in using an AspectJ ITD. Open the `ApplicationConversionServiceFactoryBean_Roo_ConversionService.aj` file and look at the static inner `AccountConverter` class that it defines. You're going to add this converter to the .java file: an easy way to do that is to use a built-in refactoring. Select the `AccountConverter` in the .aj file, right-click and choose "AspectJ Refactoring -> Push In..." and press OK.

Then switch back to the `ApplicationConversionServiceFactoryBean` and update the `convert` method to simply return the name of the account.

Make sure to also add a `registry.addConverter(new AccountConverter())` call to the `installFormatters`

method, since Roo will remove that line from the ITD after you push in the static converter class: without this call the converter won't be used!

Repeat this push-in and edit procedure for the `BeneficiaryConverter`. FIXME: doesn't currently work, AspectJ complains about empty body of installLabelConverters.

Tip

If the push-in refactoring wizard doesn't show the BeneficiaryConverter after selecting it, then cancel and perform a Project -> Clean first. Sometimes the AspectJ plugin seems to get confused after the first push-in, cleaning the project will fix that.

Restart the server and verify that your changes are reflected.

Note

Since we're using an in-memory database you'll have to enter a new account and new beneficiaries.

15.2.2.6.2. Step 10: Edit the generated views

It would be nice if you could add new beneficiaries to an account directly from the account details page. This page is rendered by the `WEB-INF/views/accounts/show.jspx` file. Open it and look at its implementation: as you can see, custom tags are used to render the account details. This ensures that changes to the entity will automatically be reflected in the user interface. However, you're free to edit this file and add some contents.

Copy the `<form:create>` element and its content from the `views/beneficiaries/create.jspx` file and paste it under the `<page:show>` element in the account's `show.jspx` file. Also copy the `xmlns:form` attribute from the create.jspx root element to the root element of the show.jspx to declare the tag library's namespace.

Notice that this tag requires a `beneficiary` model attribute for binding. To ensure that that attribute exists, add the following method to the `AccountController`:

```
    @ModelAttribute
    public Beneficiary newBeneficiary() {
        return new Beneficiary();
    }
```

Also, replace the `<field:select>` tag with the following tag, since we know the account id for the new Beneficiary already:

```
<input type="hidden" name="account" value="${account.id}"/>
```

Restart the server, add a new Account and try to add a new Beneficiary to it from the account details view using your new form. It's not the perfect solution yet, but this gives you a pretty good idea of how Roo's custom tags help you to quickly customize the generated user interface, or to define new pages yourself while still allowing Roo to generate some of the dynamic contents for you.

Chapter 16. web-test: Functional And Performance Web Application Testing

16.1. Introduction

In this lab you will automate the process of verifying a web application's functionality from end to end (as opposed to one method at at time). You will also create performance tests to see how well your application works under load.

What you will learn:

1. How to run tests with the Selenium Core Test Runner
2. How to record a test with Selenium IDE
3. How to start JMeter and use it as a proxy to record tests
4. How to model different user scenarios in JMeter
5. How to create performance tests for Web Flow in JMeter

Estimated time to complete: 75 minutes

16.2. Instructions

The instructions in this lab are divided in two sections. In the first section you use Selenium Core and Selenium IDE to record and execute end-to-end functional tests. In the second part you will use JMeter to record and invoke performance tests and to measure the performance of the application.

16.2.1. Create Functional Tests With Selenium

Before recording any tests let's take a look at Selenium Core and Selenium IDE.

16.2.1.1. Selenium Core

Navigate to `src/main/webapp` where you'll see the Selenium Core distribution. Selenium Core needs to be installed on the same website as your application, because of JavaScript's "Same Origin Policy" intended to

prevent the loading of scripts from multiple domains. You will use Selenium Core to executing functional test cases and test suites recorded in HTML files. To look at a sample test suite, open `src/main/webapp/tests/TestSuite.html`. You will see a table containing a single "dummy" test case called "Foo Test". Let's look at how we can load it in the Selenium Test Runner.

Click and drag the project to the SpringSource tc Server. Open a Firefox browser and enter the path to the Selenium Test Runner:

```
http://localhost:8080/webtest/selenium-core-1.0.1/core/TestRunner.html
```

You should see the following:

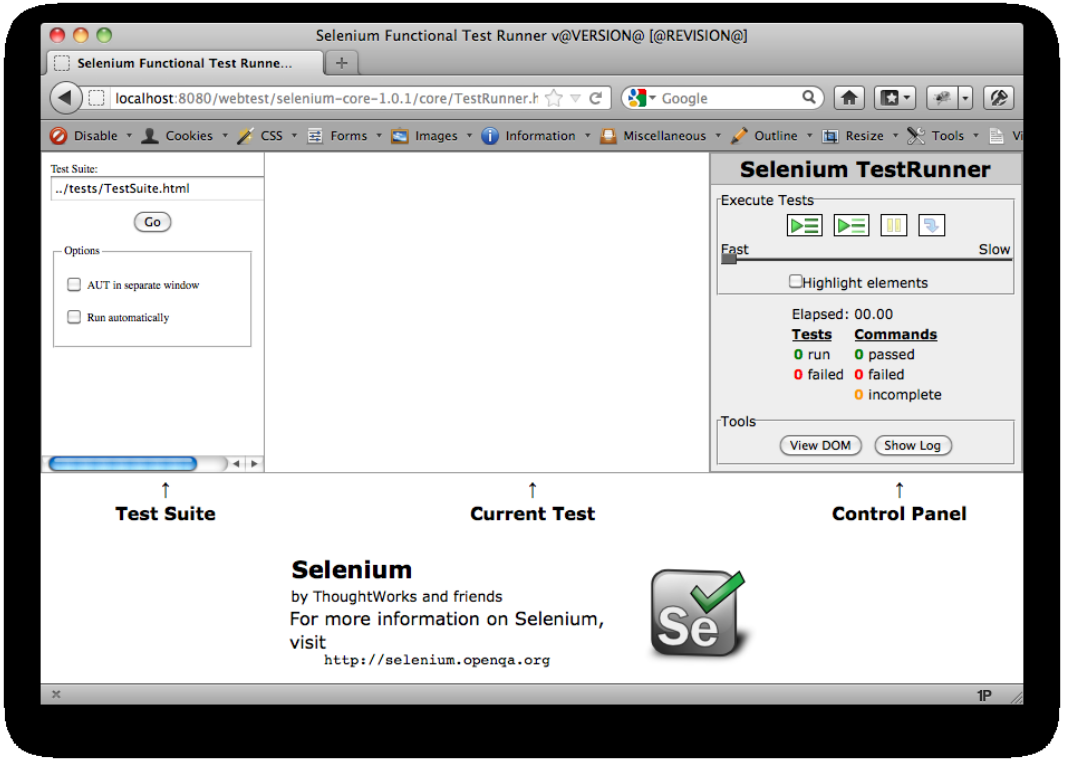


Figure 1: Selenium Test Runner

This is the Test Runner. In the top left corner you can enter the path to the test suite relative to the URL of the Test Runner. Enter the following path: `../../tests/TestSuite.html` and press Go. This should load the test suite with "Foo Test" but since the test doesn't exist a 404 error will appear in place of the test case. Now let's take a look at the Selenium IDE, which can be used to record tests.

16.2.1.2. Selenium IDE

In the Firefox browser, open a new tab and enter the path to the application:

```
http://localhost:8080/webtest
```

Next click on the Tools menu, and pick Selenium IDE. You should see the Selenium IDE:

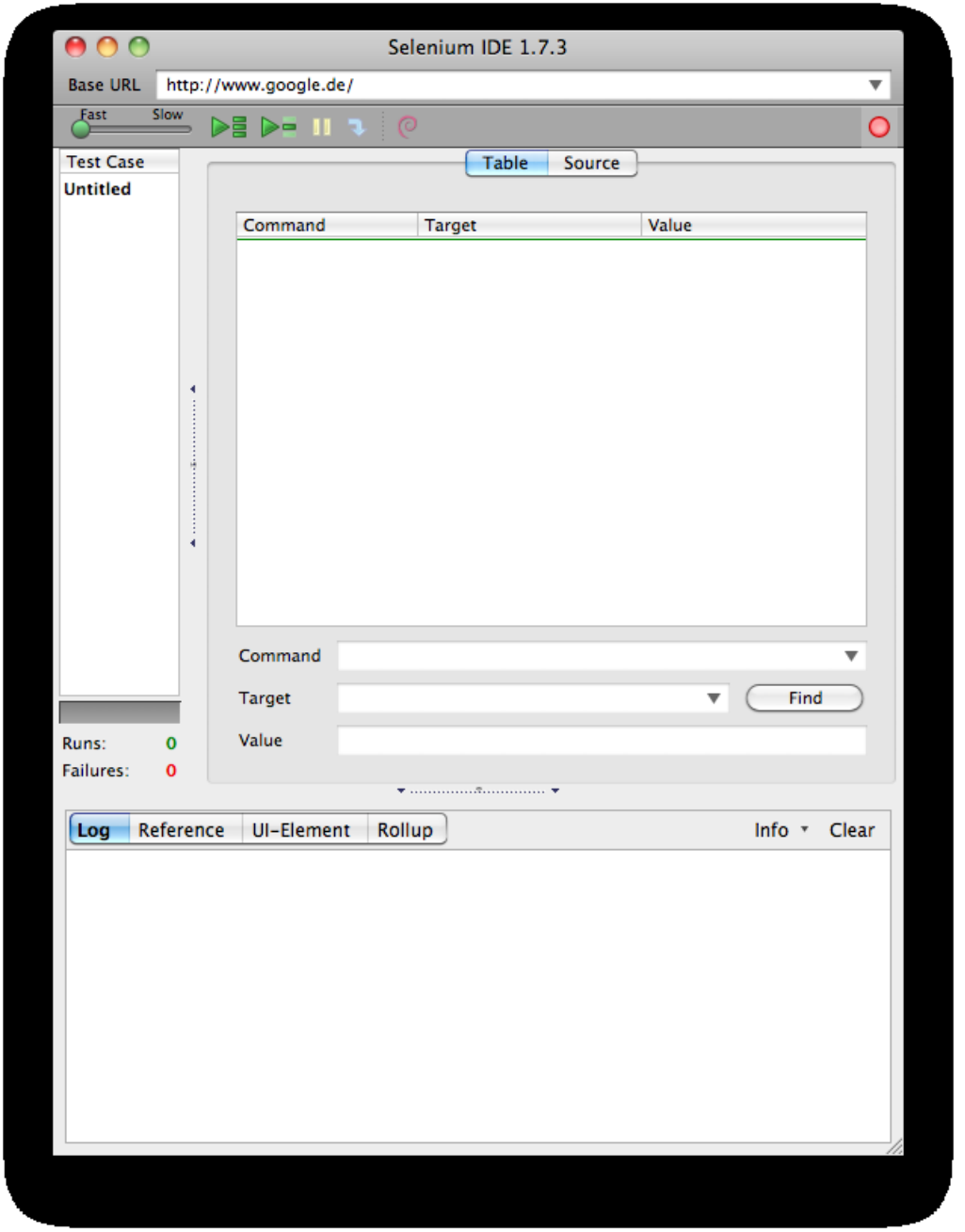

Selenium IDE 1.7.3
Base URL
http://www.google.de/
Fast
Slow
Test Case
Untitled
Table
Source
Command
Target
Value
Command
Target
Find
Value
Runs:
0
Failures:
0
Log
Reference
UI-Element
Rollup
Info
Clear

Figure 1: Selenium IDE

You will use Selenium IDE to record and save functional tests as HTML files.

16.2.1.3. Create a Functional Test For Searching Accounts

By default when Selenium IDE it is in recording mode. To verify that find the red record button on the right and move the mouse over it. You should see a tooltip that says "Now Recording". If not click it to begin recording.

Back in the main browser window click on Accounts. Notice how Selenium IDE recorded two commands: "open | /webtest" and "clickAndWait | link=Accounts". Open means open the specified URL while clickAndWait means click on the link with the specified value and wait for the page to reload.

Tip

While recording feel free to look at the values Selenium IDE creates. However if you click on one of the commands keep in mind the next thing you do will insert new commands at that point. Typically you'll want to scroll down to the bottom below the last command to make sure new steps are added to the end.

Also at any point while recording you can right-click the recorded commands and select "Clear All" to start over.

Now that we're on the Account Search page let's add an assertion. Right-click on Account Search and choose "assertTitle Account Search". Look at Selenium IDE to see that it added the appropriate assert command. You should add at least one assertion whenever a new page is loaded to guarantee the successful transition.

Next enter 'a' in the Search String field and press Search. That will take you to the search results. Since this is a page reload add another assertion, this time for the Search Results page. When done right-click "Next" at the bottom and choose "verifyElementPresent nextUrl" to verify the presence of that link.

Tip

Unlike an assertion, a verification does not fail the test case but simply records the error. It's common to create a single assertion per page and then follow up with a number of verifications. This is a nice balance between stopping a test vs. recording as many errors as we can without stopping.

Following the same step as for the "Next" link, add a verification for the "Change Search" link. Lastly right-click at least one name of the results (e.g. "Dollie R. Adams") and select verifyTextPresent Dollie R. Adams. We have sufficiently verified the current page and are ready to move on.

Click Next to get to the next page. As before, add assertions for the page title and for the name "John C. Howard" in the search results.

Tip

If we were using Ajax to load the results you would typically point Selenium to some part of the page that will be refreshed via Ajax and tell it what to look for. For example you might, right-click the first name in the results "John C. Howard" and choose "waitForTextPresent John C. Howard".

Why do this? If you don't do this, the Test Runner may execute the steps faster than it takes for the Ajax request to complete, causing the test to fail. That is why you need to tell Selenium to wait and what to wait for.

Note that it is *not* necessary to do so in this test, however, as we don't do any Ajax loading at this time.

Lastly click on the link for the details of "John C. Howard", and then press the red button in Selenium IDE to end the recording session.

Use the "Play Current Test" green button to play the entire test sequence one more time and make sure it runs without errors. Assuming it does use the File menu to save the test case to

```
 C:\spring-web-<version>\spring-web-<version>\web-test
\src\main\webapp\tests\SearchAccounts.html
```

The test has now been recorded and saved and is ready for execution with the Test Runner.

Go to the STS workspace, find the `src/main/webapp/tests` directory, and refresh it. It should contain the SearchAccounts.html file you saved from Selenium IDE. Next open `TestSuite.html` and replace Foo Test with the newly recorded Search Accounts test. When this is done go to the Firefox browser, select the tab with the Selenium Test Runner, and refresh the page. You should now see the Test Suite with the Search Accounts test:

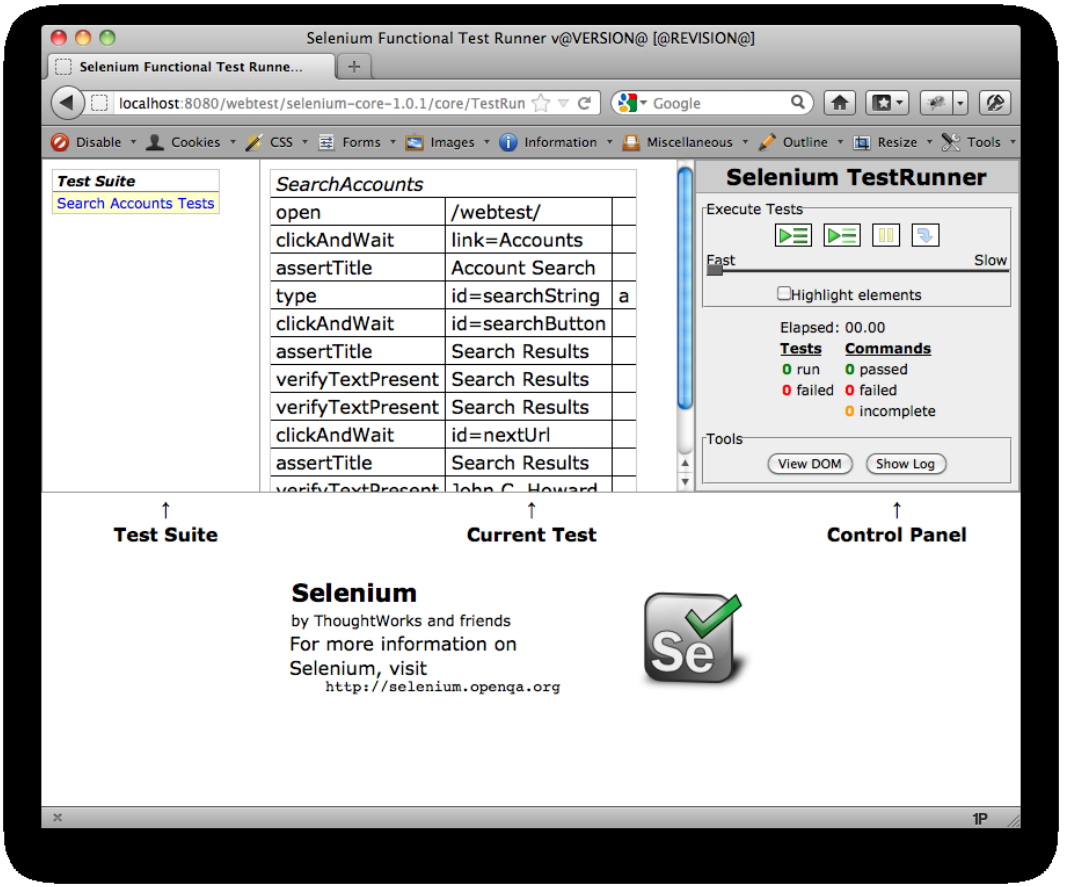


Figure 1: Selenium IDE

Press the Run All Tests button in the top right corner and watch the Test Runner execute the tests.

16.2.1.4. Create a Functional Test For Editing an Account

The process for creating the next functional step is very similar. See if you can create it on your own. Here is a sample sequence of steps:

1. Click on Accounts

2. Search for 'Keith'

3. Press Edit

4. Modify the account data

5. Save

Tip

Use 'File' - 'New Test Case' in Selenium IDE to record the new test

When you are done recording and verifying the test case save it in the same directory where the other test was saved. Add the new test case to TestSuite.html, go back to the browser tab with the Selenium Test Runner, and refresh the page. The suite should now contain two test cases. You can execute them if you like. They should run successfully.

16.2.1.5. Create a Functional Test For New Rewards

Lastly use Selenium IDE to record a test for creating a new reward. The steps should be apparent by now. After completing this step you're ready to move on and create some performance tests as well.

16.2.2. Create Performance Tests With Apache JMeter

Apache JMeter can be used to simulate a heavy server load and to analyze overall performance under different load types. It is possible to simulate different usage scenarios and it is important to create tests that are as close to real application traffic as possible. For example you can use server request logs to analyze production traffic before coming up with a performance test plan. In this lab we will assume there are 3 usage scenarios - browsing accounts, editing accounts, and creating new rewards. You will soon see how to vary the load amount for each usage scenario.

16.2.2.1. Start JMeter

Go to `c:\spring-web-{version}\jmeter-<version>\bin` and start JMeter using `jmeter.bat`. You should see an empty screen like this one:

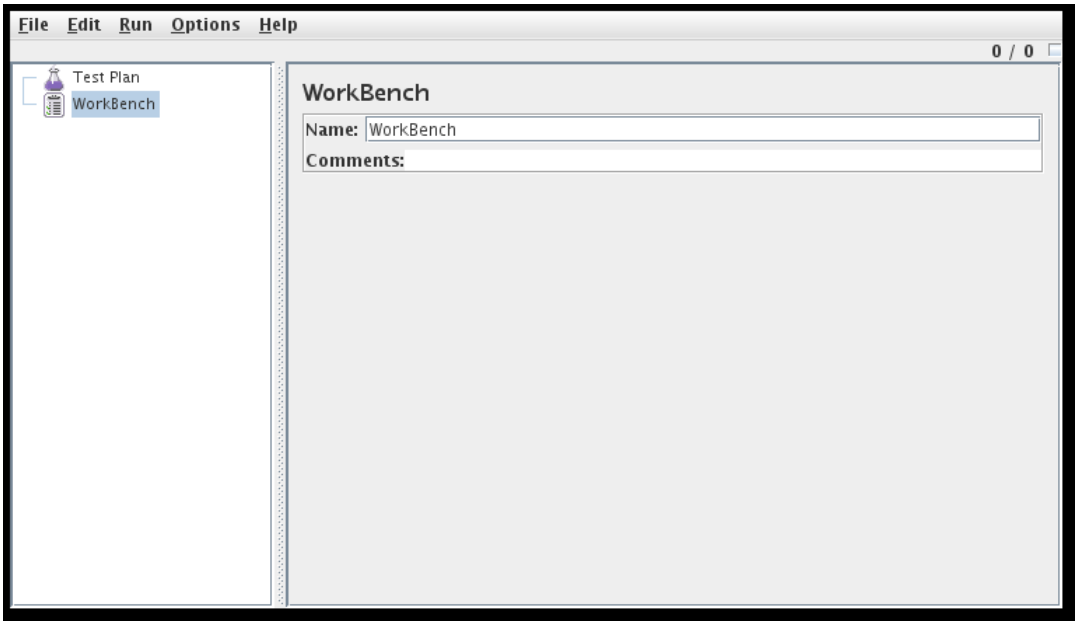


Figure 1: Default Jakarta JMeter screen

16.2.2.2. Create a Skeleton Test Plan

A test plan is the root element of a JMeter project. It is common to begin filling it out by adding one or more Thread Groups. You can think of Thread Groups as categories of users, each one using the application in a specific way.

Right-click Test Plan on the left, select Add, and then Thread Group. Click on the newly created thread group and change its name to "Browse Accounts Thread Group". Create two more thread groups called "Edit Accounts Thread Group" and "New Reward Thread Group".

16.2.2.3. Prepare to Record Tests with JMeter

JMeter can act as a browser proxy and record requests while you're using the browser. This is a very convenient way of building up a test plan. Let's set up to use it.

First we need to tell Firefox to use a proxy. In Firefox go to the Tools menu and select Options. In the preferences menu choose Advanced and then press the Network tab. Click the Settings button and then set up a manual proxy for localhost and port 9090 (Be sure to remove localhost from the field labeled "No Proxy For" if it is there!)

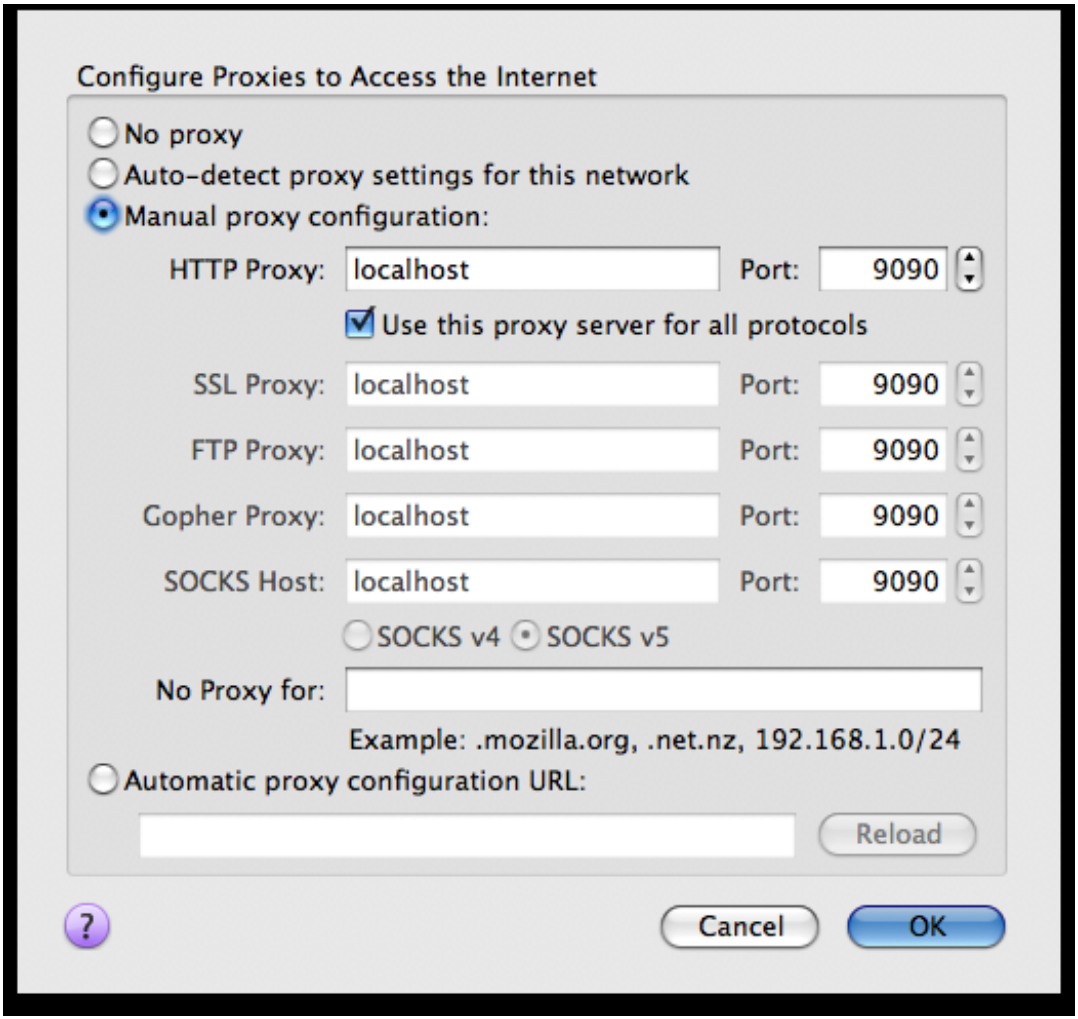


Figure 1: Firefox Advanced Network Preferences

Now Firefox will send all requests through localhost:9090 but they will fail unless there is an actual proxy at that address.

Return to JMeter, right-click WorkBench (on the right-hand side), select Add - Non-Test Elements - HTTP Proxy Server. Set the port to 9090 and set the "Target Controller" to the Thread Group you created. Next, press the Add button below "URL Patterns to Include" and type the pattern .* to make sure all requests are recorded. Next, press the Add button below "URL Pattern to Exclude" and add .*.css, .*.js, .*.png, and .*.gif. Your settings should be similar to the ones on the picture below.

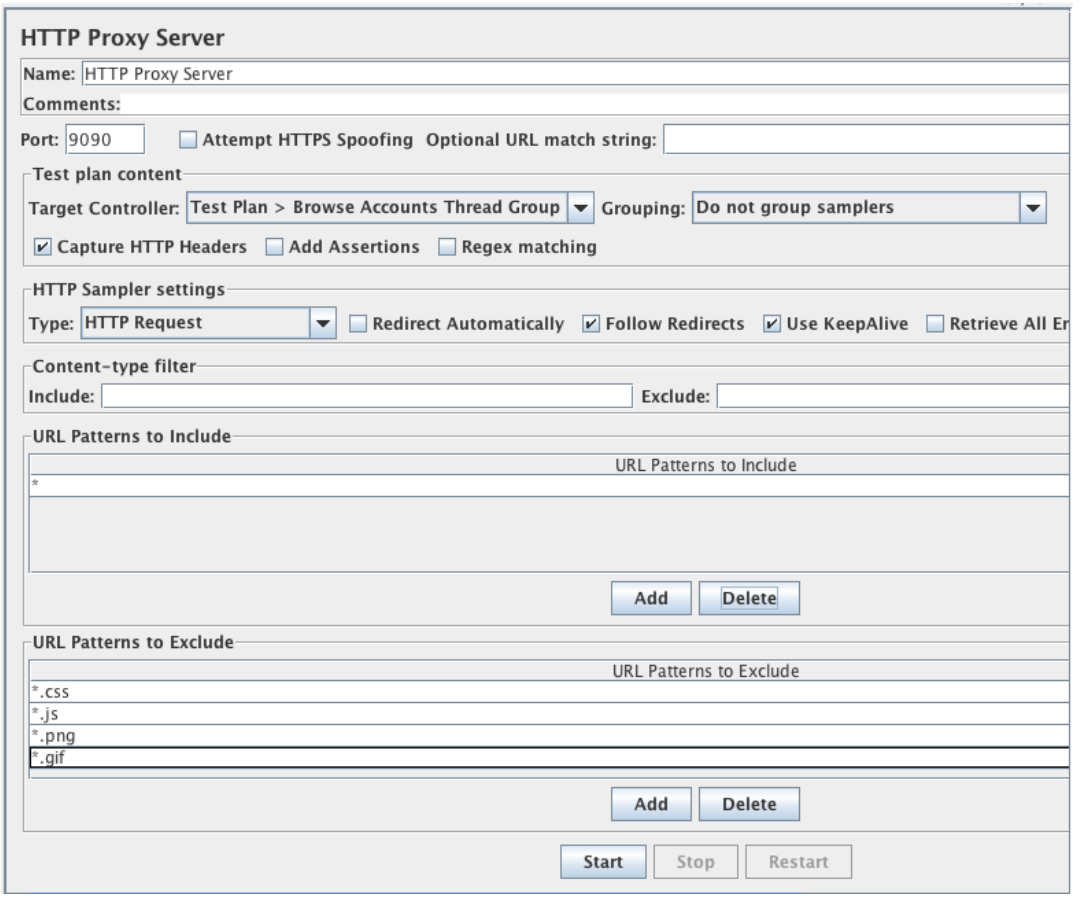


Figure 1: JMeter HTTP Proxy Server Settings

You are now ready to start JMeter's proxy. Use the Start button at the bottom to start it.

Browse a few pages of the application in the browser, return to JMeter and make sure they've been recorded as steps under the "Browse Accounts Thread Group". Once you've verified recording works move on to the next step.

Tip

If you don't see anything getting recorded, try closing and reopening the Test Plan. This will force JMeter to refresh its contents.

16.2.2.4. Record a Performance Test For Browsing Accounts

Select all steps recorded so far and remove them through the right-click menu. The real fun is about to begin. Just remember you can always remove any recorded steps and repeat them in the browser as necessary.

In Firefox click on Accounts, enter 'a' in the search string field and press Find. Go back to JMeter. See the two recorded requests. Right-click the "Browse Accounts Thread Group" and add a logic controller of type Random Controller.

Tip

Logic controllers allow you to exercise recorded steps in different ways such as by looping over requests, interleaving them, invoking them in a random order, and so on.

Select the second recorded request (the one doing the actual search) and use Ctrl+X to cut it. Select the Random Controller item and use Ctrl+V to paste into it three times. You now have three instances of an account search request executed in a random order. Click on each request and change the searchString parameter to 'a', 'john', and 'keith' respectively. Finally click on Random Controller and rename it to "Search Random Controller".

In Firefox click on Accounts, enter 'a' in the search string field and press Find. On the search results page press Next once and return to JMeter. Create a second Random controller under "Browse Accounts" and call it "Pagination Random Controller". Cut the last recently recorded link and paste it 3 times into "Pagination Random Controller" (you can discard the other recorded steps). You now have three instances of requests paging through search results executed in a random order. Click on each pasted link and change the "page" parameter to 1, 2, and 3 respectively so they're each going to a different page of the result set.

In Firefox click on an account to navigate to the Account Details page. Use the browser back button and visit a few more accounts. Return to JMeter and add all newly created steps under a 3rd random controller called "Details Random Controller".

To make things more interesting and more realistic add an Interleave Controller directly under the "Browse Accounts Thread Group" and then cut and paste the 3 random controllers making them children of the Interleave controller. What you now have is a Browse Accounts thread group, which mixes (via interleaving) random requests for searching, paging, and showing account detail pages.

16.2.2.5. Execute a Load Test

Click on the Browse Accounts thread group, change the number of threads (users) to 30 and set the loop count to Forever. Based on these settings the load test will start up to 30 concurrent requests adding one each second (the ramp-up period) and looping until you tell it to stop.

One more thing is left before running. We need to be able to view the results. Right-click on Test Plan and add a listener of type Aggregate Report. Make sure Aggregate Report is the currently selected item and then go to the Run menu and select Start. As the tests ramp up the report will begin showing the results. The test will continue running until you stop it through the Run menu. Feel free to do so at any time and then examine the final results. You will see various numbers such as minimum and maximum time for a request, the median, the average, and the 90% line. Just keep in mind that average can be a very misleading number and it's usually better to watch the median or the 90% line to get an idea of what most users experience. Throughput is another important performance metric.

Now that you're done with the Browse Accounts thread group, right-click and disable it before moving on to the next test.

16.2.2.6. Record a Performance Test for Editing Accounts

Go to the HTTP Proxy Server under Workbench and change the target controller to "Test Plan > Edit Accounts Thread Group". This will ensure recorded steps will now appear under the Edit Accounts thread group.

In Firefox navigate to an account, make a change to it and save it. Use the browser back button to go back and select another account. When you've edited 3 or 4 different accounts you can stop. That's all we need for this group.

Return to JMeter and click on "Edit Accounts Thread Group", change the number of threads (users) to 20 and the loop count to Forever. Use the Run menu to launch the load test for a bit. After a little while stop the test and check the results. When you're satisfied right-click and disable this thread group before moving on to the final step.

16.2.2.7. Record a Performance Test for New Rewards

By now you've hopefully gotten the hang of creating JMeter tests. There are two unexpected issues that arise when recording JMeter tests against Web Flow.

The first issue is that Web Flow is stateful and needs to store values in the HTTP session. By default servlet containers store the jsessionId in a cookie but since we're simulating users from the same machine we also want many different cookies. Fortunately JMeter has built-in support for that. Right-click "New Reward Thread Group" and add a config element of type "HTTP Cookie Manager". The Cookie Manager ensures each simulated client request doesn't have to share its cookies with any other requests (seems like a good idea any time we talk about any kind of cookies).

The second issue has to do with dynamic parameters such as Web Flow's execution key. It changes from request to request and cannot be hard-coded in the test scripts. To deal with this in JMeter we need to use a Post Processor element to parse a server's response, extract the dynamic value and parameterize subsequent requests with it.

Tip

This is a problem we did not encounter with Selenium. Unlike JMeter Selenium runs as a bot inside the browser and does not need to record actual request parameters. Instead Selenium records user actions such as typed values, mouse clicks, and others. It then lets the browser submit additional dynamic values such as Web Flow's execution key.

Let's illustrate how to record a Web Flow test by example. In Firefox click on New Reward. Return to JMeter and verify there are two requests. The first is the New Reward link, the second is a client-side redirect containing the execution parameter.

Tip

Remember that by default Web Flow does a client side redirect on every request.

Right-click the first request and add a new Post Processor of type "Regular Expression Extractor". Its settings should look like this:

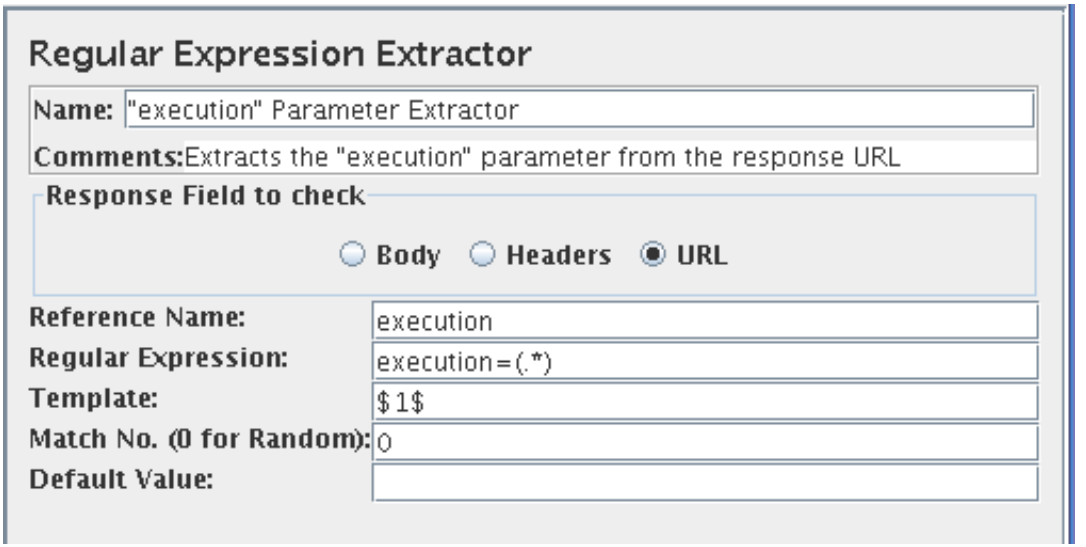


Figure 1: Regular Expression Extractor

This post processor uses a regular expression to extract the "execution" parameter value from the server response, saves it as a variable called "execution" so it is then available for use to subsequent requests. Click on the second request and modify the hard-coded execution value with ${execution}.

Tip

In terms of the actual regular expression, (.*) creates a single grouping. The template 1 then accesses that grouping. That's what the value of execution is set to.

In Firefox fill out the dining information, press Reward and return to JMeter. Click on the 3rd request and insert another Regular Expression Extractor. The extractor should have identical settings as the one above. Click on the fourth recorded request and replace the hard-coded execution value with ${execution}.

Tip

Note that if this call had been an Ajax request, you would additionally set the "Response Field to check" to Headers.

When an Ajax request results in a client-side redirect, Web Flow returns a normal HTTP response (200) but sets an HTTP response header field called "Spring-Redirect-URL" to the actual redirect URL. A JavaScript handler provided by Spring detects that and decides how to handle it.

In Firefox press the Confirm button to complete the flow and return to JMeter. Click on the last request and notice it needs a confirmationNumber. This is also a dynamic value and cannot be hard-coded. Click on the previous request, insert yet another Regular Expression Extractor similar to the first ones but this time extracting a confirmationNumber parameter. When this is done click on the last request and replace the hard-coded confirmationNumber value with ${confirmationNumber}.

Finally, check the URLs for the GET requests in this flow. Anytime an execution parameter is included, you will need to replace it with the dynamic ${execution} parameter.

You are now done preparing the New Reward test script. Run the test once through the Run menu with a single user and check the Console in STS to make sure there are no exceptions. If you see any Web Flow exceptions they might be because of incorrectly extracted parameter values. If you don't see any exceptions click on the "New Reward Thread Group", set the number of threads to 10 and the loop count to Forever.

Click on the other two thread groups and enable them. Click on each group one at a time and verify the number of threads (users). You should have 30 account browsing users, 20 account editing users, and 10 creating new rewards. You're welcome to change these values as you wish. When ready run the tests one last time, all of them together. Remember to monitor the results through the Aggregate Report item. In addition to Aggregate Report there are also other listener and post-processor elements that allow recording and showing results in different ways. Feel free to experiment if you have any time left.

Congratulations, you've now completed this lab.