VALIDATION DSL FOR CLIENT-SERVER APPLICATIONS

VALIDATION DSL FOR CLIENT-SERVER APPLICATIONS

BY

VITALII FEDORENKO

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTING & SOFTWARE

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

© Copyright by Vitalii Fedorenko, July, 2012

All Rights Reserved

Master of Science (2012) McMaster University
(Computing & Software) Hamilton, Ontario, Canada

TITLE: VALIDATION DSL FOR CLIENT-SERVER APPLICATIONS

AUTHOR: Vitalii Fedorenko (vitalii.fed@gmail.com)

SUPERVISOR: Dr. Jacques Carette

Abstract

Given the nature of client-server applications, most use some freeform interface, like web forms, to collect user input. The main difficulty with this approach is that all parameters obtained in this fashion need to be validated and normalized to protect the application from invalid entries. This is the problem addressed here: how to take client input and preprocess it before passing the data to a back-end, which concentrates on business logic. The method of implementation is a rule engine that uses Groovy internal domain-specific language (DSL) for specifying input requirements. We will justify why the DSL is a good fit for a validation rule engine, describe existing techniques used in this area and comprehensively address the related issues of accidental complexity, security, and user experience.

Acknowledgements

I am thankful to my supervisor, Dr. Jacques Carette. The discussions I had with him and the comments I got were always helpful and informative. I am grateful to my wife, Anna, for her inspiration, love, and understanding. A special gratitude to my parents, who instilled in me a lifelong passion for learning.

Table of Contents

1 Introduction..1
1.1 Background..1
1.2 Roadmap..2
2 Input Validation..4
2.1 Validation on server and client sides..4
2.2 Sanitization 5
2.3 Type check..6
2.4 Canonicalization..7
2.5 Business logic validation..8
2.6 Tamperproofing..8
2.7 Example of input validation..8
3 Common problems in input validation..11
3.1 Essential and accidental complexity..11
3.2 Accidental complexity of validation code..12
3.3 User experience..13
3.4 Security..14
3.5 Lack of code reuse..16
4 Current Techniques for Input Preprocessing..18
4.1 View level validation..18
4.2 Field level validation..21
4.3 Form level validation..23
4.4 Model level validation..25
4.5 Defining rules using monads..30
4.6 Business Rules Management Systems..32
4.7 JavaScript validation..32
4.8 Sharing of validation code on client and server side..34
4.9 Leveraging Ajax for input validation..35
4.10 Web application firewall..36
4.11 Summary..36
5 Framework Requirements..38
5.1 Centralized input preprocessing..39
5.2 Whitelist validation..40

5.3 Cross-site request forgery and protection from bots 41
5.4 Expressive language syntax..42
5.5 Client side...43
5.6 Library of common converters and validators......................................44
5.7 Summary...44
6 Input Data Flow 46
6.1 Form validation...46
6.2 Validation of navigation request..50
6.3 Summary...51
7 Choice of Groovy DSL...53
7.1 Pros and cons of Groovy DSL..53
7.2 Other considered options..55
8 DSL Syntax and Semantics...56
8.1 Rules and functions scripts...56
8.2 Validators...57
8.3 Converters..57
8.4 Rule application...58
8.5 No-operation converter...58
8.6 Composite rules 59
8.7 Optional parameters...60
8.8 Type conversion...61
8.9 Accessing other parameters values...61
8.10 Rule groups 62
8.11 Closures...63
8.12 Combined parameters..64
8.13 List parameters...64
8.14 Custom parameters..65
8.15 Conditional rules..65
8.16 Default converters and skip function..66
8.17 Reusing rules...66
8.18 Rule extension...67
8.19 Multi-line rule...67
8.20 Boolean converters...67
8.21 Variables...68
8.22 Not logged rules...68
8.23 Validation block..69
9 Script Management..71
9.1 Script inclusion..71
9.2 Grails plugin...72
9.3 Rule engine API...72
10 Error Handling...75
10.1 Basic error handling...75
10.2 Custom error parameters..76

10.3 Validation exception....76
10.4 Redirect URL....77
10.5 Value placeholder....78
10.6 Handling of grules exceptions in main application....78
10.7 Handling of non-validation exceptions....79
11 Client Library....80
11.1 Initialization....80
11.2 In-line validation....81
11.3 Client validation events....83
11.4 Form submission....84
11.5 Validate function....85
11.6 Managing of validation events....85
11.7 W3C standards support....86
11.8 Summary....86
12 Configuration....87
12.1 Configuration file format....87
12.2 Properties....87
12.3 Logging....88
13 Built-in Functions....89
13.1 Validators....89
13.2 Converters....94
14 Example....98
15 Conclusion and Future Work....102
15.1 Conclusion....102
15.2 Future work....103
Appendix....104
Core rules script grammar....104
Bibliography....105

Chapter 1

Introduction

1.1 Background

There are two main types of data with which a client-server application operates: internal and external. The former consists of values from a local environment — constants, functions computation results, parameters from a configuration file, a secure database etc. This data is protected from outer changes and usually requires only trivial checks, such as handling of null pointers, verification of type boundaries and structural constraints imposed by a data model. Internal data is expected to be safe unless it was modified intentionally by a person or a program that has access to the application codebase at compile or run time. On the other hand, data sources which can contain foreign data, like user supplied input, result of an external API call, or records from a shared database, are external relative to the main application and should be considered to be potentially invalid (accidentally or otherwise), and thus need to be checked in accordance with some set of more complex rules.

The most common validation issues that client-server application developers must deal with are incorrectly filled forms. The basic flow of a registration process in web applications, for instance, begins with a client requesting a page with a form, filling in user information, and clicking the "Submit button" to send the data to a server for processing. The data from such a form is not much use if the would-be user does not type in a username or email, for example. In addition, an application might require that the values it receives from a form are in the correct format — a name contains only alphabetic characters and a URL for a personal website is a well-formed resource locator. Finally, we should check that the age is within certain boundaries.

If not handled properly, data from untrusted sources could conceivably alter a normal application flow or even give an attacker illicit access to your site [12]. Thus, any external value should be thoroughly analyzed by an application and assumed to be malicious until proven otherwise. After the data has been tested the system can accept or reject the input and alert the user to existing problems. From the above, we can say that the purpose of an all-encompassing validation system is firstly to

ascertain whether a data set contains the expected content and each parameter value has been properly formatted, while at the second stage the input has to be sanitized against acceptable content specification and validated by business rules before updating the application environment.

In the past decade, several validation frameworks have appeared with the aim to solve this problem, but we have found that most of the existing solutions have issues with expressivity, security, and user experience. In this document we present a solution that mitigates these problems by the creation of a rule engine with declarative, functional DSL implemented in Groovy. In our work, we leverage best practices in security, design, web development, and user experience.

Though there are many academic research papers on specific methods of checking user input, we could not find one that clarifies common ways to validate input in web projects that would be supplemented by criticism of each approach, emerging best-use patterns, and recurring anti-patterns, so we tried to describe some of the most popular techniques in this document.

It is assumed that the reader is familiar with the basics of Java (or, better, Groovy), JavaScript, HTML, regular expressions, and the Model-View-Controller pattern [42]. Some examples are provided in Scala, Python, Haskell, PHP, C#, and XML, but proficiency in these languages is not essential for understanding the underlying concepts.

1.2 Roadmap

The rest of this document is structured as follows:

Chapter 2, Input Validation, discusses how input data is preprocessed on the client and server sides, which validation stages the data needs to pass and why.

Chapter 3, Problem Description, describes the current state of affairs in preprocessing of data in online applications and provides an overview of the problem that we want to solve.

Chapter 4, Current Techniques in Input Preprocessing, is about the approaches currently used by frameworks in various languages to validate data from the client side.

Chapter 5, Framework Requirements, derives the features a framework should have to at least partly solve the existing problems in processing of external data.

Chapter 6, Input Data Flow, gives a step-by-step description of the proposed process of validating data on the client and server sides.

Chapter 7, Choice of Groovy DSL, evaluates Groovy as a host language for internal DSL and explains why it was chosen over other available alternatives.

Chapter 8, DSL Syntax and Semantics, defines the language grammar and meaning of supported rule expressions.

Chapter 9, Script Management, explains how scripts can interact with each other and specifies an API that can be used to directly access functionality of the rule engine.

Chapter 10, Error Handling, shows the syntax for specifying errors in the rules script and introduces an error handling mechanism available on the side of the host application.

Chapter 11, Client Library, covers a client-server communication protocol and callbacks exposed to a client for adjusting the validation process to custom UI requirements.

Chapter 12, Configuration, provides information about the format and content of the framework configuration file.

Chapter 13, Built-in Functions, specifies a list of validators and converters available in the provided functions library.

Chapter 14, Example, demonstrates how the framework can be used in an online application to preprocess input from a web form.

Chapter 15, Conclusion and Future work, contains a summary of our results, discusses the strengths and weaknesses of the implemented approach, and presents ideas for future work.

Chapter 2

Input Validation

As alluded to above, to ensure that the application is robust against all forms of input, whether obtained from the user, infrastructure, or database systems, all external data should be checked according to business requirements and security policies. These requirements should rigorously constrain what values are allowed — which types, patterns, ranges, and any entries that do not match these constraints must be rejected. In this case, the main application logic can assume that the data satisfies all expected properties and invariants.

User input has to be validated both on the server and on the client (generally web browser), consequently, we have server and client side validation. In this chapter we will take a brief look at features peculiar to each of these stages and define five main types of input preprocessing: well-formedness, sanitization, canonicalization, business-logic validation, and tamperproofing.

2.1 Validation on server and client sides

When a user submits a filled out form, the entered data is packaged in a request and transmitted to the server, where the submitted values are checked for syntactic and semantic correctness before any business logic is applied. This point in the request/response cycle is called server-side validation. For instance, before adding a new account to a database, an application can test that an email address has the right format and that the desired username is available. If an error is found, a response with the corresponding information is sent back to the client and shown in a form understandable by the user, usually by displaying an error message next to a problematic form field.

If only server side validation is implemented, users receive feedback about the input data only after it has been processed by the server. This results in additional time spent to enter the data, as periodic round-trips (termed postbacks) to the web-server

are needed. While some postbacks are unavoidable, you should always be mindful of ways to minimize travel across the wire, especially for environments where the Internet connection is unreliable or expensive (like mobile clients). One technique that saves postbacks is using client side scripting to validate user input before submitting the form data to the web server [111]. Client side validation provides the user with immediate feedback as he types or submits a form, and if the input is in the wrong format, not all required information is supplied, or there is some other validation problem, the client script notifies the user with an error message and does not allow to post back to the web server until the error is corrected. In this way, it improves usability with a more responsive interface, reduces the time spent filling in the form and communication traffic [82].

In web applications client side validation is handled by code in a programming language that can be run within a browser environment (such as JavaScript, Dart [124], etc). However, it can be easily turned off by a user, which is the main drawback of this type of validation. Additionally, even if the script has checked the data, there is always the possibility of modifying a request that was formed by a client application before it gets to a server. This is why a client should never be trusted, and validation must be implemented on both sides — otherwise there is no guarantee that the application will collect valid data. By using double validation, we gather the best features of the two: fast response, more secure validation and a better user experience [51]. Also, since there is often a part of validation checks that cannot be carried out by a browser (like user credentials), the server side set of validation rules is always a superset of the client side rules.

Another technique often used by developers of online applications is Ajax technology (Asynchronous JavaScript and XML). It allows to instantly validate entered data on the server without reloading a page. The pros and cons of Ajax will be discussed further in Chapter 4.

2.2 Sanitization

One of the first stages in input preprocessing is sanitizing, which can be defined as removing accidentally or deliberately added redundant symbols and making potentially malicious data safe. This stage should be done with utmost caution, as without sanitization, an attacker can compromise the application by supplying carefully crafted input to attempt SQL injection [101], cross-site scripting (XSS) [93], abnormal URI input [102], or another type of attack [98]. For example, in cross-site scripting, an attacker can exploit vulnerabilities in input validation by injecting client side script code in the HTTP parameters values, and when the script code is embedded in the response, a user's browser evaluates it as part of the page. With

this technique, one can easily steal authentication cookies or other secure data stored on the client side, and since the browser downloads the script code from a trusted site, it has no way of recognizing that the code is not legitimate (unless sandboxing technique is used) [99].

There are two main strategies for sanitization of external data: via whitelist and via blacklist. In whitelist sanitization, any character which is not a member of an approved list is removed, encoded, or replaced [34]. For example, if a phone number is required for input, all non-digit characters that occur in it have to be stripped out. Thus, `(555) 123-1234`, `555.123.1234`, and `5551231234\";DROP TABLE USER;--` will be converted to just `5551231234`.

On the other hand, in blacklist sanitization an input and output are analyzed for a set of known characters (or sequence of characters) that *do not* meet the application security policy or can lead to data corruption. In an effort to make the input safe for further processing, such characters are eliminated or escaped so that they will be treated as literals and never as, for instance, an interpreter instruction. The blacklist approach is often impractical as it has overhead costs for maintaining the set of patterns for unacceptable data, which can be relatively large and change over time. For example, there are 70 unique ways to encode the "less than" symbol and in this way inject an HTML tag for XSS attack [100], so while some services prohibit special HTML symbols like `<`, `/`, and `>` in the user input, these characters sometimes can be encoded as `%20`, `%2F`, and `%3E` respectively to circumvent a sanitization filter. Nevertheless, when it is possible to prove that all possible variations of bad input are well-known, will remain finite, and the range of valid data cannot be defined in advance (so we cannot match on a whitelist), adopting the blacklist strategy can still provide good protection.

Sanitization has to be performed for all application tiers: on the presentation layer, an application should check values against XSS attacks, on the persistence layer — against SQL injection, for directory lookups — for LDAP injection [94], and so on. This allows a database library or a template engine to make certain assumptions about the content of free variables which values are supplied at runtime.

2.3 Type check

A type check validation ensures that a provided input value is of the right primitive type, such as an integer, double, or character. For example, at this stage an application may check that the specified age is an integer value. This check is generally implemented using regular expressions or type converters. Violation of data

type requirements makes further processing of the entry impossible.

A common mistake at this stage is direct casting from an initial data type to a target primitive type without additional format checks. Such implementation can result in run time exceptions or other undesirable application behavior.

2.4 Canonicalization

Data in canonical form is in its most standard or simplest form, while canonicalization is the process of converting data to such a form [92, 104]. To establish the canonical form of the input, this process takes into account current character encodings, locale, and other system settings [31]. For example, the following strings specify the same file path in Linux and share the same canonical representation — an absolute file path to this file (`/home/myuser/tmp/file.txt`):

```
tmp.txt
~/tmp/file.txt
~/tmp/./file.txt
/home/myuser/tmp/file.txt
~/tmp/subdir/../file.txt
```

For such a canonical form to exist there should be a surjective function `f` that converts any valid initial value to a canonical form, and another function `g` that in turn bijectively maps the canonical form to entities under consideration (to satisfy this requirement both `f` and `g` should always terminate).

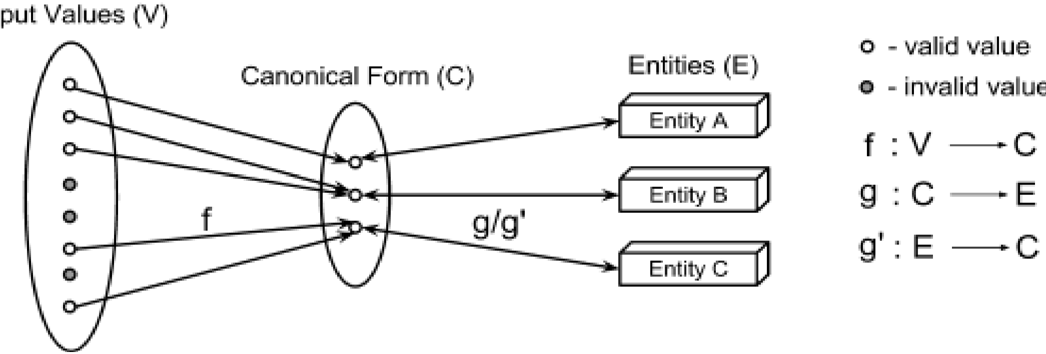

Figure 1: Canonicalization

Web applications are particularly prone to issues pertaining to incorrect canonicalization, as many well-known exploits use different representations of file paths and URLs to get unauthorized access to system resources. Thus, such data

should be always transformed to a canonical form before it is processed by an application.

2.5 Business logic validation

Business logic validation is a process that ensures data is sane in terms of the application model — that numbers are correctly signed and within limited range, strings have the right length, a desired username is available and so on. That is, it prevents cases where a value type is valid, but the business rules behind the application disallow this concrete value. This form of validation usually occurs after both the type check and sanitization have already been applied.

A simple example would be a shopping cart limitation in an online store — at the type check stage, a quantity of an ordered product is parsed to an integer, while business-logic validation tests that the user has not exceeded his credit limit. For string values it is convenient to implement business-logic validation using a regular expression language. For instance, if an application expects a US zip code, the regular expression `\d{5}(-\d{4})?` can be used.

It also should be mentioned that validation cannot be considered as a way to bring a user to enter true (or at least useful) information, but only to ensure syntactically and semantically valid input. For example, if you allow a name field to contain only alphabetic characters and a site visitor tries to input fake data like an arbitrary number of digits, it would be unreasonable to expect that after seeing an error message about the wrong format, he will enter his real name instead of "qwerty" or any other dummy alphabetic string.

2.6 Tamperproofing

On the server side, an application needs to be always aware that after some value is transferred from a server to a client it may be changed by the latter. Integrity checks (or tamperproofing) are essential to ensure that in the case of using data provided by a client that was earlier retrieved from the server, it has not been tampered with between client/server interactions. Tamperproofing must be included wherever data passes from a trusted to a less trusted boundary, such as from a web server to a user's browser in a hidden field or cookie [34]. For example, the integrity check can inspect whether a client uses the same transaction ID on all steps of a money transfer form.

2.7 Example of input validation

Suppose we develop a bank application in which a funds transfer transaction is performed in two steps. Initially, the server responds with the form that contains the `<input>` field for the amount to be transferred and the `<select>` field with values for possible payee accounts:

```
<form method=”post”>
  <input type=”text” name=”amount”/>
  <select name=”payeeid”>
    <option value="1">University Account Department</option>
    <option value="2">Internet Service Provider</option>
    ...
  </select>
</form>
```

While filling in the web form, the user chooses the recipient and sends the selected value to the application server. The first step the application has to make regarding the recipient ID is to check the value against security policies (*sanitization*) and ascertain whether it has an integer type (*type check*), then the ID is checked for existence (*business-logic validation*). Similarly the server performs validation for the `amount` field, but also converts its value to the format with a fixed number of decimal digits (*canonicalization*). When this stage is finished successfully, the second form with other fields and the transaction ID as a hidden input is sent back to the user. After the user has filled it in and clicked the submit button, the transaction ID with all other values is sent back to the server where the ID is checked for integrity (*tamperproofing*). For example, the following pseudo code validates the transaction ID parameter and retrieves the corresponding recipient account information:

```
if ‘transactionId’ in http parameters:
  transactionId = sanitize http parameter ‘transactionId’
  if transactionId is not a natural number:
    throw InvalidTransactionIdError
  if <transactionId, payerId> in currentTransactions:
    recipientAccount = get recipient by transactionId
  else:
    throw NoSuchTransactionIdError
else:
  throw NoTransactionIdError
make transfer to recipientAccount
```

In this example we described only one possible invocation order of validation stages, but in general, it can vary for different types of inputs. The sole restriction is that

business-logic validation should be always applied to values first processed by other stages.

Chapter 3

Common problems in input validation

A common scenario used to implement user input validation for a client-server application includes the following steps:

1. At first, to make sure that the data processed by the business layer will be sane, a developer implements validations and transformations for each parameter upon the necessity of accessing it in a form controller on the server side.

2. After that, if the input can be further used in a presentation layer or as a database record, sanitization rules are implemented as a countermeasure to server attacks. In addition, tamperproofing may be used when the value is automatically embedded as part of input for other forms.

3. Now, to improve user experience, he will reflect part of the validation logic in form handlers on the client side using JavaScript, Dart, Objective-C, or some other programming language.

4. Finally, the developer writes test suites for the validation code for both the client and server side.

Though imperative style validation described above is quite intuitive and easy to implement, code created through such a process makes the system more complex and implicitly compromises security, as well as one of the most crucial parts of any application that involves human interaction — user experience. In this chapter we will dwell upon each of these aspects in detail.

3.1 Essential and accidental complexity

First of all, let us define two types of code complexity [16]:

Essential complexity is inherent in, and the essence of, the problem solved by the application (as seen by the users). In the case of validation code, the essential complexity depends only on input constraints imposed by business and security requirements.

Accidental complexity is complexity non-essential to the problem, and with which a developer would not have to deal in the "ideal" world (e.g. that is arising from performance issues, suboptimal design, language or infrastructure).

The use of a general purpose programming language for some problems can increase accidental complexity of the code because of the absence of semantics that can express the business logic in an easily comprehensible way, and we believe that validation systems implemented in such languages provide a good example of this.

3.2 Accidental complexity of validation code

In most client-server applications, you will find that input values are accessed in form controllers via a simple key-value map and canonicalized/validated in the beginning of request processing or just before applying the business logic that depends on these parameters. As you will see in this chapter, the validation process often involves satisfying requirements from different layers, since a specially formed parameter value can pass business logic validation but be malicious if inserted into a database or an HTML response. Typically, it splits the implementation of the input validation across different modules, raising its intellectual overhead. Furthermore, we think that it is also undesirable for a business logic to validate and convert input values, as it undermines the single responsibility principle [105]. These factors cause lack of uniformity in application design, which reduces clarity of the validation model and negatively impacts comprehensibility of the whole system. This complication quickly becomes evident when the application's codebase begins to grow and commonly goes along with the state explosion problem [67].

If one wants to improve user experience by adding form validation to the client side, he usually needs to implement the same preprocessing rules that are run on the server, except in JavaScript, Dart or some other programming language. Further, he should synchronize the code for all platforms each time requirements for the input are changed. This certainly breaks the DRY rule (Don't repeat yourself) [81] and

means that there is no single "source of truth". For large projects, such violations induce error-prone conditions and in a long-term perspective, can make the system difficult to maintain.

Yet another problem with this common way of processing user input is that since imperative languages like Java, PHP and Python are now dominant in a web domain, the rules are usually not so expressive as they can be in declarative representation (expressivity in this context refers to the measure of how readily, concisely, and humanly understandable the logic can be specified [95]). Besides the redundant verbosity this deficiency introduces to the code, imperative style hampers code generation facilities that can be used for automatic emission of validation handlers for the client side. And while some frameworks provide an XML or JSON [3] syntax for defining the validation rules, because of the limitations in semantics of these languages, they can be used only for simple cases. Moreover, use of markup languages for the rules declaration raises some integration issues which reduce application maintainability and testability.

Finally, it is worth noting that by nature, validation code is not particularly inspirational to write. It can be exciting to develop functionality that dynamically generates charts or adds some social features, but it is hard to imagine a coworker boasting of the interesting way she stopped a user from entering a blank value for a name field [43]. Therefore, we expect that a more agile process for the writing of tedious validation code should improve the quality of client-server applications.

As an example of the accidental complexity problem you can study the source code of ThinkUp, which is the most followed PHP application on Github. You may find that since there is no systematic approach to preprocessing of input data in ThinkUp, and because of this some operations are applied on request parameters without type check, which makes results of evaluation of such expressions prone to errors. The other example is the MediaWiki [125] project, a core engine of Wikipedia: while its `WebRequest` class provides a check for primitive types, all other validations of HTTP parameters are performed by controllers themselves. For example, each time a request parameter is used, it is manually checked for existence. Such design makes it difficult to reason whether all parameters were properly validated before any business logic was applied. Also, MediaWiki "solves" the problem of code duplication by validating input only on the server side that evidently negatively affects user experience.

Combination of all of these factors can notably increase accidental complexity of the code while reducing its quality and we expect that elimination of the issues mentioned in this section can in many ways simplify application design.

3.3 User experience

Complexity begets further complexity. Consider a form with poor client-side validation and imagine a casual visitor trying to go through the registration process using this form. At first, the potential user of the service fills it out and submits the input data to the server. However, after a couple of seconds she gets the same page thrown back in her face and finds that some values are not filled correctly, the username is already taken, required fields are missing, another password should be created because the provided one is not strong enough, and, to top it all off, even the correct values should be selected again as the fields that contain them were reset to defaults. In some cases, after the failed submit a frustrated user might postpone the registration process or even leave the site, especially if the web site has competitors that offer a similar service or if it is not so crucial for the user; for those who do proceed with registration, the first impression may be damaged. A possible solution is to show detailed instructions on how to fill out each field, but in practice users tend to skip such information and thus still become annoyed by a rejected input.

According to UX specialists, site input validation is one of the top problems in web usability issues, and in our opinion, the main reason is the current complexity of validation systems.

3.4 Security

As opposed to general software engineering, in security, what you overlooked is more important than what you implemented. Usually, if you understand the basic syntax and semantics of the given programming language, you can implement almost any logic given a sufficient amount of time, system specifications, and relatively mild performance and quality requirements. However, if something was missing in your security system, consequences can be irreversible. If an application makes any unfounded assumptions about the content, type, length, format, or range of input, it can easily become a security hole. When discovered by an attacker, such false assumptions may help game application business logic or interpret malicious code on the server to get unauthorized access to site resources.

One of the main security problems in input validation is that, without a single checkpoint for all external data, occasionally a parameter might be validated differently in two places. In the worst case, it may even "slip" through business rules to a database or presentation layer, which assume that each free variable satisfies

some properties, like being a literal of an expected type. The other risk is that new kinds of vulnerabilities appear from time to time and keeping validation rules up to date with all types of attacks requires a special skill set and incessant time investments. And while gaps in preprocessing of external data have become the most common web application security weakness, only a minor part of industrial software engineers has sufficient background in security to perform verification covering all possible ways of misrepresenting data [91]. From personal experience, developers often do not take into consideration character encoding features such as canonical equivalence [11] and multi-byte sequences [126], or even tend to assume that HTTP parameters which come from navigation links and hidden inputs will be not altered. Also, sometimes input sanitization that seems trivial at first sight can be substantially more intricate. For example, a naive developer can try to remove scripts from HTML code just by stripping the `<script>` tag and `onEvent` attributes, but might miss tricky cases like these:

```
<img src=x:alert(alt) onerror=eval(src) alt=0>
<b/alt="1"onmouseover=InputBox+1 language=vbs>test</b>
```

The other type of malicious exploit of a website is cross-site request forgery (CSRF) [84]. It allows the attacker to trick a user into performing an action using his authority and credentials by sending a POST HTTP request from the client browser to a target web application. When a form is submitted from a malicious site, the victim's browser automatically sends the authorization information stored in the cookie with the request. If the victim is logged into the site, data submission is performed as the web application does not know that this was a forged request. To monitor such service misuse, detection of a request origin should be a part of form submission preprocessing. Because of operational scope limited to string processing, validation frameworks rarely provide an interface for implementing CSRF protection, and separate intrusion detection systems (IDS) designed for this purpose are usually used (for example of IDS see [28]).

There are also some automated tools, such as web robots, developed with the unique purpose of undertaking illegal activities or achieving goals that are not in line with the ethic of web utilization. By imitating typical Internet-surfing human behaviour, such as filling out a form, the robots can perform a series of actions violating usage policies of online services [47]. Without control over such requests, a server becomes vulnerable to security threats and spam, therefore easing their detection is critical to site safety.

Unfortunately, in consideration of human imperfection, software peer review cannot guarantee that all security flaws will be found, and no code checker can ever assure us that validation is done properly. To be usable, such a tool should be sound and complete, which is not yet feasible as the checker should be aware of all business

requirements and understand how the data is used on each application layer (a complete code checker would find all errors, while a sound one would report only real errors and no false positives [40]).

The other commonly applied mechanism used to to gauge software security is penetration testing. Passing software penetration tests may prove that no faults occur under particular test conditions, but by no means does it prove that no faults exist. Misunderstanding this point is one of the main problems with today's approaches to finding software vulnerabilities [88]. In practice, you will discover that to provide the best possible security a combination of the described methods is used, though even then you cannot guarantee that an application is immune to attack.

If we take a look at the statistics, Gartner Group says that 75 percent of hacks are at the web-app level and, that out of 300 audited sites, 97 percent were vulnerable to attack [12], while the WhiteHat Website Security Statistics Report notes that 64 percent of three thousand analyzed sites have had at least one serious security hole [110]. This shows that protection from hacker attack is one of the hardest parts of web development and advanced security techniques should be used to shield a product from intrusion.

3.5 Lack of code reuse

One of the most widely used tools for validating input are regular expressions that constrain entry format to some pattern. Nearly every developer has, at some point, written his own regex for a postal code, URL, or an email address. Although such "in-house" rules are usually acceptable for the simplest cases, they are also frequently prone to reject some exceptional valid value or miss an invalid input. As proof of this statement, let's take the example of validation of an email address — a field present in almost all registration forms. Few software engineers know that the local-part of an address can be quoted with any ASCII characters in between the quotes [24], or that the plus symbol (`+`) can be used to add a certain label to the message [52] (i.e. an email to `me+spam@gmail.com` is actually sent to `me@gmail.com` but a recipient will see the label `spam` next to the title). In order to check if such characters are handled correctly by web applications, we tested the top twenty sites by world traffic and found that only two of them accept quotes while more than half do not expect the plus sign in an email address. While it is always useful to detect as many ill-formed values as possible, when it comes to entries such as a Social Security Number (SSN), not many web sites check if a given number can actually exist [113]. Another example is a regular expression for alphabetic symbols in languages other than English: if one tries to filter such characters using the `[a-z]` pattern in Russian (`[а-я]`) and some other alphabets, the regex engine can skip some letters from the

pattern [89], which is obviously not intuitive behavior and can cause bugs in the program. There are also examples that can cause serious security issues, like an escape function or parser for a file path. Even such a simple mathematical function as calculating an absolute value in Java can cause troubles if used unadvisedly as it returns a negative value for `Integer.MIN_VALUE` (so if you use it, for example, for conversion of a number of a particular product in a cart, the total price of an order can be a negative value).

There are innumerable amounts of similar fine points for other common types of entries (see [112] for a further example). Moreover, the rules can change with time or some additional data may be required to conduct comprehensive validation (zipcodes, city names etc). Therefore, the existence of a uniform validation system that takes into account such subtleties and manages communication with the data source would undoubtedly simplify development of web applications and make them more solid.

Chapter 4

Current Techniques for Input Preprocessing

This chapter discusses a number of techniques used to improve plain imperative style validation. All of them substantively differ in the ways and degrees with which they resolve the problems of code expressivity, security, complexity, and user experience, so we will provide individual analysis for each solution.

4.1 View level validation

One of the most intuitive ways to make the validation model more expressive is to explicitly bind the constraints for input values to form elements in HTML markup. This technique was adopted in ASP.NET Web Forms, JavaServer Faces [1, 107], and some other frameworks [44]. Here is an example in ASP.NET that describes how to add a validator that will make a user name input be a required field:

```
<form action=”/login.asp” id=”loginForm”>
  <table>
    <tr>
      <td>User name:</td>
      <td>
        <!-- Input field -->
        <input type=”text” runat=”server” id=”usrName”/>
      </td>
      <td>
        <!-- Validator with error message -->
        <asp:RequiredFieldValidator runat=”server”
            ControlToValidate=”usrName”
            ErrorMessage=”User name is required.” />
      </td>
    </tr>
```

```
    <tr>
      <td colspan="3">
        <!-- Submit button -->
        <input type=”submit” value="Submit" name=”submit” id=”submit”
            runat=”server”/>
      </td>
    </tr>
  </table>
</form>
```

And the same form in JSF:

```
<f:view>
  <h:form id="loginForm">
    <table>
      <tr>
        <td>User name:</td>
        <td>
          <!-- Input field with validator -->
          <h:inputText id="usrName" required="true"
              requiredMessage="User name is required" />
        </td>
        <td>
          <!-- Error message -->
          <h:message for="usrName" style="color:red" />
        </td>
      </tr>
      <tr>
        <td colspan=”3”>
          <!-- Submit button -->
          <h:commandButton id=”submit” value="Submit"
              action=”#{loginBean.login}” />
        </td>
      </tr>
    </table>
  </h:form>
</f:view>
```

Having validation logic close to an input element declaration has its strong and weak points. On the one hand, it keeps the object and subject of validation (a field and its server handler) in one place, which makes comprehension of the data preprocessing flow easier than in models that define validation rules separately from an input form. On the other hand, it leads to duplicated code — with this method of validation, you need to copy the rules in all of the forms. While this disadvantage can be insignificant for prototypes or web sites with a relatively small number of pages, for other projects it is time consuming and leaves you prone to future errors, as in

this case, when making any changes to your business rule logic, it is easy to forget to update every instance. Furthermore, this approach can make application design more complex; for instance, to create a custom validator or converter in JSF, one needs to register it in several configuration files, then, if multiple fields are used, a backing bean, the backing bean class, and finally, an extra hidden field must be created to conform with the framework API. Due to limitations of the declarative style, combining two or more validators in a non-linear composition requires defining of a new validator with all appropriate beans. Last but not least, there is no simple way to reuse the same approach for HTTP parameters in navigation links, cookies, or headers, which should be processed as well as form inputs.

The main origin of these problems is the violation of the principle of separation of concerns. Mixing of validation logic and views causes high coupling of the business and presentation layers, in particular, form fields become cluttered with validation code — which is really metadata about the input data itself. Because of it, any framework that uses view-level validation will have similar problems and solving them would require rethinking its input data model in toto.

In some web frameworks, form elements and the corresponding validation rules are both defined using the syntax and semantics of a host language. Django [73] — a framework for developing web applications in Python — is one of the representatives of such a technique, as shown in this example of a registration form:

```
class RegistrationForm(forms.Form):

  """Input for the username"""
  username = forms.CharField()

  """Input for the password"""
  password = forms.CharField(widget = forms.PasswordInput())

  """Input for the password confirmation"""
  confirmPassword = forms.CharField(widget = forms.PasswordInput())

  def clean_password(self):
  """The method is called for password validation"""
    password = self.clean_data.get('password', '')
    confirmPassword = self.clean_data.get('confirmPassword', '')
    if password != confirmPassword:
      raise forms.ValidationError("Passwords do not match")
    return password

form = RegistrationForm()
# Generate the form
form.as_table()
```

Today it is also used in WebDSL, an experimental domain-specific language for the development of web applications [60]. The following listing shows how the same registration form can be validated in WebDSL:

```
define page register(u: User) {
  var confirmPassword: Secret;
  form {
    group("User") {
      label("Username") { input(u.username) }
      label("New Password") { input(u.password) }
      label("Re-enter Password") {
        input(confirmPassword) {
          validate(u.password == confirmPassword, "Passwords do not
              match")
        }
      }
      action("Save", action{ })
    }
  }
}
```

Unfortunately, this approach has the same issues as “pure” view level validation, with the only difference that now the rules can be easily defined using the syntax and semantics of the main application language, but at the same time, the customization of the form becomes a more difficult task.

4.2 Field level validation

To deal with issues intrinsic to embedding validation constraints into the view, some alternative techniques were developed. One of the ideas is to separate the flow of the application from the web pages. Struts [29] is probably the most popular Java framework that implements this strategy by using field level validation. This type of validation consists in specifying a separate rules file that defines validness criteria for each input parameter. In Struts, these requirements are written in a domain-specific language based on XML, while custom validators should be implemented in the Java language. For example, to mark an input parameter as required and to check that its value is an integer within the range of 10-20, you need to write a snippet similar to this one:

```
<field property="integer" depends="required,integer,intRange">
  <arg0 key="typeForm.integer.id"/>
  <arg1 name="intRange" key="${var:min}" resource="false"/>
```

```
  <arg2 name="intRange" key="${var:max}" resource="false"/>
  <var>
    <var-name>min</var-name>
    <var-value>10</var-value>
  </var>
  <var>
    <var-name>max</var-name>
    <var-value>20</var-value>
  </var>
</field>
```

An interface of the custom validator should be also defined in the XML format:

```
 <validator name="equals"
   classname="com.javasrc.struts.validators.StrutsValidator"
   method="validateEquals"
   methodParams="java.lang.Object,
       org.apache.commons.validator.ValidatorAction,
       org.apache.commons.validator.Field,
       org.apache.struts.action.ActionMessages,
       org.apache.commons.validator.Validator,
       javax.servlet.http.HttpServletRequest"
   depends="" msg="errors.equals"/>
```

While field level validation isolates business logic from a presentation layer, the way it is implemented still has some critical issues, like inexpressive syntax and lack of handy integration with controllers [56]. The suggested DSL imposes a high level of syntactic noise in rule definitions, though, its basing on XML allows to some degree automate the process of emitting code for validation on the client side. Also, defining a custom validation rule in Struts requires modification of the form controller and the configuration file, in addition to adding some JSP code to display validation error messages back to the user. As such, validation is best added during the initial design, as including the validation model into existing code requires changing inheritance trees and could easily break the application [27].

XML is not the only format used for defining field validation rules in web applications. Codeigniter [74] — a framework used in building sites with PHP — uses string based rules with its own format. For example, to define a cascading rule in Codeigniter one can pipe multiple rules together with the vertical bar:

```
$this->form_validation->set_rules('username', 'Username', 'trim|
required|min_length[5]|max_length[12]|is_unique[users.username]');
```

With string literals, you can define any kind of syntax for validation rules as long as you can parse it in a rule engine, but this resilience does not come for free: you lose

compile time verification, IDE support, and possibility to inline custom validators using the language the application is written in. Starting from PHP 5.2, you can use built-in validation utilities, but they are limited to string filter functions that only support processing of primitive types, emails, IP addresses and URLs [66].

4.3 Form level validation

In form level validation, rules for each form are defined in a separate class which encapsulates all validation logic. These class methods are called by a controller to check input parameters before any business logic is applied to their values.

Since in this approach the validation schema is defined using the syntax and semantics of the application language, in addition to declarative definitions, one can imperatively express dependencies between form fields and the model, which adds much more flexibility in comparison to the previously described field level validation. Let us give an example from Symfony [85], a web application framework written in PHP. At first, we will describe validation rules for the contact form fields and a custom validator for the name of a department:

```
class ContactForm extends sfForm {

  protected static $subjects = array('SubjectA', 'SubjectB');

  public function configure() {
    $this->setValidators(array(
       // a subject value is from the predefined set $subjects
      'subject' => new sfValidatorChoice(array('choices' =>
          array_keys(self::$subjects))),
      // a message size is more than 4
      'message' => new sfValidatorString(array('min_length' => 4)),
      // a department is checked by the department_valdiator function
      ‘department’ => new sfValidatorCallback(array('callback' =>
          array(array($this, 'department_valdiator'))
    ));

    $this->validatorSchema->setPostValidator(
      // a start date is before the end date
      new sfValidatorSchemaCompare('start_date',
          sfValidatorSchemaCompare::LESS_THAN_EQUAL, 'end_date',
      array('throw_global_error' => true),
      // the error message
      array('invalid' => 'The start date ("%left_field%") must be
```

```
        before the end date ("%right_field%")')
      )
    );
  }

}

/**
 * Validator for the department name field.
 */
public function department_valdiator($validator, $value) {
  if (is_department($value)) {
    return $value;
  } else {
    throw new sfValidatorError($validator, 'Error message');
  }
}
```

Then we can add a call to the `sfForm.isValid()` method to the contact form controller:

```
class ContactActions extends sfActions {

  public function executeIndex($request) {
    $this->form = new ContactForm();
    if ($request->isMethod('post')) {
      $this->form->bind($request->getParameter('contact'));
      if ($this->form->isValid()) {
        // a redirect to the success page
        $this->redirect('contact/thankyou?'.http_build_query(
            $this->form->getValues()));
      }
    }
  }

}
```

A careful reader will note that Symfony uses a combination of imperative and declarative paradigms for defining validation rules. Among other advantages, this allows framework plugins to parse the declarative part of the schema and automatically add client side validation to form inputs [86].

Similarly to Symphony, in ASP.NET MVC 3 you can express validation rules both with imperative code and field attributes. This can be seen in the following class that validates input from the contact form:

```
public class ContactMessage :IRuleEntity {

  public string Author { get; set; }

  [RequiredValidator("You must enter a message subject.")]
  public string Subject { get; set; }

  [RequiredValidator("You must enter a message body.")]
  public string Body { get; set; }

  public List<RuleViolation> GetRuleViolations() {
    var validationIssues = new List<RuleViolation>();

    // Validate Author in imperatively
    if (String.IsNullOrEmpty(this.Author)) {
       validationIssues.Add(new RuleViolation("author", this.Author,
           "You must enter an author name."));
    }

    // Validate all attributes
    AttributeValidation.Validate(this, validationIssues);
    return validationIssues;
  }

  ...

}
```

Here, the `Subject` and `Body` properties in the `Message` class are decorated with the `RequiredValidator` attribute which marks them as required, and at the same time the `Author` attribute is validated imperatively in the `GetRuleViolations` method.

Using a separate class for form declaration allows to reveal fields that are not properly checked and thus eliminate potential security holes. However, embedding declarative definitions into an imperative language implies constraints in the rules' expressivity, and, as you may have noticed, the definitions can become too verbose and do not allow to easily compose or in-line validators.

4.4 Model level validation

The alternative approach to considering validation rules as part of presentation layer would be to define the constraints in relation to an application model and then bind

input parameters to the corresponding fields of data layer entities. Subsequently, to check input data, the system can run the validators on each model object that has input parameters bound into it and ensure that all of them have succeeded.

Some popular Java libraries implemented model level validators leveraging annotations introduced in JDK 1.5. Most of the frameworks, including Hibernate Validator [58] and Seam [63], are based on JSR 303 [57] that defines a metadata model and API for entity validation. The Spring Bean Validation Framework [59] would be another example that allows to perform validation declaratively using the annotations, but formally it does not adopt the full specification. Here is the typical Spring form bean:

```
public class User {

  @NotBlank
  @Length(max = 40)
  private String name;

  @NotBlank
  @Email
  @Length(max = 40)
  private String email;

  public String getName() {
   return name;
  }

  public void setName(String name) {
   this.name = name;
  }

  public String getEmail() {
   return email;
  }

  public void setEmail(String email) {
    this.email = email;
  }

}
```

Then in the JSP file you need to define form inputs and where validation errors will be displayed:

```
<form:form commandName="user">
```

```
  <table>
    <tr>
      <td>Name:</td>
      <td>
        <form:input path="name" size="40” cssErrorClass="err-fld"/>
      </td>
      <td>
        <div class="err-msg"><form:errors path="name"/></div>
      <td>
    </tr>
    <tr>
      <td>Email:</td>
      <td>
        <form:input path="email" size="40” cssErrorClass="err-fld"/>
      </td>
      <td>
        <div class="err-msg"><form:errors path="email"/></div>
      </td>
    </tr>
    <tr>
      <td colspan=”3”><input type="submit" value="Submit"/></td>
    </tr>
  </table>
</form:form>
```

Now, in the controller you validate user input via `Validator` object and then populate the model with clean data:

```
@Controller
public final class ContactController {

  @Autowired
  private Validator validator;

  public void setValidator(Validator validator) {
    this.validator = validator;
  }

  @RequestMapping(value = "/form", method = RequestMethod.POST)
  public String post(@ModelAttribute("user") User user,
      BindingResult result, Model model) {
    validator.validate(user, result);
    model.addAttribute("user", user);
    if (result.hasErrors()) {
      return "form";
    }
```

```
    return "redirect:thanks";
  }

  @RequestMapping(value = "/form", method = RequestMethod.GET)
  public ModelMap get() {
    return new ModelMap("user", new User());
  }

}
```

Also, each possible error message should be defined in a properties file in the following format:

```
User.name[not.blank]=Please enter your name.
User.name[length]=Please enter at least {2} characters.
...
```

Finally, you will also need to add a few beans to your Spring configuration file.

Model view validation via annotations proved to be useful for the checks that represent certain simple formatting rules, e.g. `EmailAddress`, and do not require logic operations or imperative-style expressions to specify the validation criteria. However, we have found that a process of implementing model level validation that falls outside trivial constraints is quite cumbersome and imposes a design that is not easy to maintain.

The multiple standard annotations are not adequate on their own — certain business rules, like that usernames must be unique, might relate to the interaction of multiple properties and involve arbitrarily complex logic. And while the API for defining a custom validator is well standardized by the JSR, it requires the creation of several additional Java classes and beans in the XML files. Annotations are very limited in terms of possible compositions: the only supported operation is conjunction, which requires creating a separate interface each time you want to define a validation rule that comprises several constraints. For example, the only logic implemented by the following annotation is validation of the parameter value size and non-equality to the null [71]:

```
@NotNull
@Size(min = 2, max = 14)
@Target({ElementType.METHOD, ElementType.FIELD,
     ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = {})
@Documented
public @interface ConjunctionExample {
```

```
  String message() default "{com.foo.constraints.validlicenseplate}";
  Class<?>[] groups() default {};
  Class<? extends Payload>[] payload() default {};
}
```

Due to the limitations of the Java language, application sequence of property validators is nondeterministic: the validator that occurs as the first annotation may be applied after all others. The problem can be fixed using validation groups, but we found this approach overly verbose. For example, here we define that the `NotNull`, `NotEmpty`, and `Length` validators must be applied in a strict sequence:

```
@GroupSequence({One.class, Two.class, Three.class})
public class Person {

  @NotNull(groups = {One.class})
  @NotEmpty(groups = {Two.class})
  @Length(groups = {Three.class})
  private String name = null;

  // similarly for the date field
  @NotNull(groups = {Two.class})
  @Past(groups = {Two.class)
  private Date date = null;

  public String getName() {
    return this.name;
  }

  public Date getDate() {
    return this.date;
  }

}
```

Another problem that should be addressed here is that when the fields under validation are parts of different entities in the application model, a developer's responsibility is then to manually annotate each of them with a corresponding rule even if they are bound to the same HTTP parameter. For example, if a category identifier is used in several forms across the site, it should be separately declared as a candidate for validation in each model object that contains this parameter. In addition to these issues, only a few types of literals are allowed as annotation parameters [6]. This limitation does not permit to specify dynamic behavior using annotation parameters and requires encapsulation of validation logic into a separate class if it has any free variables which values are provided at runtime.

Ruby on Rails [15] (RoR) is another framework that provides easy-to-use model level validation system. In RoR, the implementation of validation logic consists of defining functions that get called when a data model object is saved. Some of the problems you can find in the implemented mechanism are lack of static verification of the validation checks and unnecessary verbosity. However, built-in validation macros, such as testing whether an input is a number or a well-formed email address, can be declared more concisely. Since there is no similar incorporated mechanism for a validation check as part of a controller action, RoR requires wrapping input validation in a data model class and making it a part of the object-relational mapping session, which clutters the domain model with artificial entities and business logic. [2]

While some frameworks allow generation of JavaScript code for basic validation routines, we have found that integration with on-client code is usually not flexible enough and a lot of boilerplate code still should be written. Nevertheless, model view validation stays one of the most popular techniques for input preprocessing, especially in the Java world (for those who are familiar with Scala, you can find a similar approach used in the Play framework's Constraints annotation [38]).

4.5 Defining rules using monads

Monads provide many useful general-purpose features and some languages try to leverage their functional programming nature by performing validation using this data structure. The monads mechanism helps hide implementation details for the sequence of function applications which makes validation rules more readable. Consider the following example that uses Scalaz [48] — one of the most popular Scala libraries that notably extends its functional programming features:

```
def person {

  sealed trait Name extends NewType[String]

  object Name {
    def apply(s: String): Validation[String, Name] =
      if (s.headOption.exists(_.isUpper))
        (new Name {val value = s}).success
      else
        "Name must start with a capital letter".fail
  }

  sealed trait Age extends NewType[Int]
  object Age {
    def apply(a: Int): Validation[String, Age] =
```

```
      if (0 to 130 contains a)
        (new Age {val value = a}).success
      else
        "Age must be in range".fail
  }

  case class Person(name: Name, age: Age)
  def mkPerson(name: String, age: Int) =
    (Name(name).liftFailNel |@|
    Age(age).liftFailNel){ (n, a) => Person(n, a)}

  mkPerson("Bob", 31).isSuccess assert_= true
  mkPerson("bob", 131).fail.toOption assert_= some(nel1(
      "Name must start with a capital letter",
      "Age must be in range"))
}
```

Despite the fact that using the monads in Scala you can write validation rules that are more expressive than their Java counterparts, the verbosity entailed by static typing makes complex rule definitions and custom validators not very readable. In addition to human readability issue, this problem can be a barrier for automatic client side code generation, which makes Scalaz less attractive than its declarative alternatives.

A similar technique can be also found in the Haskell web library [96]. For example, the following snippet show how to validate date or time:

```
validDate (Date m d) = m `elem` [1..12] && d `elem` [1..31]
dateFull = dateComponent `check` ensure validDate "This is not a valid
date"

time = do
  hours <- rangeCheck 0 23 widthTwoNum
  separator <- optionMaybe (oneOf ";,.")
  minutes <- rangeCheck 0 59 widthTwoNum
  return (TimeOfDay hours minutes 0)
```

Using monads, we are able to validate the date and time variables in this snippet at one go, without any pattern matching or equivalent bureaucracy such as conditional statements. The gains for our simple example may seem small, but it scales up well for more complex situations as well.

Advanced support of functional programming, type inference, and monads improve comprehensibility of validation code in Haskell. The main disadvantages we see lie completely outside the language semantics area: they are unpopularity of the

language as a choice for web applications and absence of production-ready JVM or .NET implementation.

4.6 Business Rules Management Systems

Business Rules Management System (BRMS) is a software system that allows business rules (i.e. logic) to be abstracted from the application itself and put into some type of external entity, so that the business rules of an application may change without affecting the application itself. In addition to a runtime environment, the system should include a rules repository and tools, allowing both technical developers and business experts to define and manage decision logic [118]. Some of the popular solutions are Validation Application Block [114], IBM WebSphere ILOG [115], and Drools [116]. To express application business rules an external DSL is usually used. For instance, in Drools, the rules can be coded in a language called MVEL [117]. You can find the following example in the JBoss Rules Reference Guide:

```
package com.company.license;

rule "Is of valid age"
when
  $a: Applicant(age < 18)
then
  $a.setValid(false);
end
```

To implement input preprocessing similar code should be written for each validation rule, and a developer also has to create error handlers and help classes that marshal the response back to a client.

Since such rules management systems aim to operate with business rules in general, they tend to be much more universal than necessary for expressing input validation rules. This factor makes BRMS not always the right tool for implementing the data validation layer unless you are using it already. There are two other points that one should consider before using these systems: serious dependency on IDE support that limits development facilities and current absence of sufficient integration with a web-based client.

4.7 JavaScript validation

There are a number of JavaScript libraries that allow you to define validation rules for the client side in declarative fashion via JSON or some other data structure. The one we will use in this section is a popular jQuery [4] plugin by Jörn Zaefferer [77]. First, let us create an example of a simple registration form:

```
<form id="register">
  <table>
    <tr>
      <td>Name</td>
      <td><input name="name"/></td>
    </tr>
    <tr>
      <td>Email</td>
      <td><input name="email"/></td></tr>
    <tr>
      <td>Gender</td>
      <td>
        <select name="gender">
          <option value=""> -- please choose -- </option>
          <option>Female</option>
          <option>Male</option>
          <option>Other</option>
        </select>
      </td>
    </tr>
    <tr>
      <td>About yourself</td>
      <td><textarea name="about"></textarea></td>
    </tr>
    <tr>
      <td colspan="2"><input type="submit" value”Submit”/></td>
    </tr>
  </table>
</form>
```

Now we can add constraints for each of the fields:

- Name must exist and cannot have any special characters.
- Email should be in a proper format.
- Gender should be specified.
- *About yourself* field should be of a reasonable length.

With the validation plugin, that is simple to set up:

```
$(document).ready(function(){
  $.validator.addMethod("textOnly",
    function(value, element) {
      return /[a-z]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF]
/.test(value);
    },
    "Only alpha characters are valid."
  );

  $('#contact_form').validate({
    'rules': {
      'name': {
        required: true,
        textOnly: true
      }
      'email': {
        required: true,
        email: true
      },
      'gender': 'required',
      'about': {
        required: true,
        minlength: 100
      }
    }
  });
});
```

The validator is activated from the moment you click on the submit button, where it checks the form, and if validation does not succeed it sets handlers on all the form elements so that they are checked as a user types the data.

As you can see, the rules format is reasonably expressive and a developer can effortlessly write his own custom validators. However, to reuse the same rules on the server, one would need to parse the JSON declaration and JavaScript validators and then translate them into Java, Python, or some other language used on the server side. Even if this process were automated, the part of validation code that requires access to the application model entities should still be written in a main language. Finally, the plugin lacks conversion and logical operators, which requires a developer to write custom handlers where they can be avoided.

4.8 Sharing of validation code on client and server side

Some code duplication relating to validating user-input can be reduced if the application is built on top of Google Web Toolkit (GWT). GWT is a set of tools and libraries that allows web developers to create an Ajax applications in Java. Its most important component is the translator, which enables the translation of a Java program into browser independent JavaScript code [41]. GWT is a fairly promising framework but, alas, there are still some reasons why it is not the best choice for web application:

- Low performance due to the amount of code that needs to be generated.
- Integration of custom HTML, CSS or JavaScript libraries in a view is a nontrivial task if it presupposes tight communication with code generated by GWT.
- The framework implies defining markup elements in the Java language which overloads implementation of simple components with redundant verbosity.

Though GWT solves one of the main problems — code duplication — just by itself, it does not settle all other problems described in the previous chapter like code complexity and security issues.

Thinking from the other direction, one might be tempted to use the client side language on the server side. For example, node.js is a JavaScript based framework, that, amongst other things, allows sharing part of the application code on both sides so most validation rules need not be written twice. However, as JavaScript was designed for handling HTML pages and not as a general purpose programming language, it lacks some crucial features which makes development of enterprise applications less convenient.

4.9 Leveraging Ajax for input validation

Ajax is one of the key underlying concepts behind Web 2.0 which lets you use JavaScript to send and receive information from the server asynchronously without page redirection or refreshing, and then partially update the page. Using server side generated data in the same fashion as it was taken locally allows to improve user experience, as site visitors are not confronted with waiting for full-page reloads or a

blank form as a reply on submit. This gives a more natural feel of an application, and in the context of input validation, reduces the amount of code duplicated on both the client and server sides.

In spite of its extensive usage Ajax still has some disadvantages:

- A round-trip to a server is required. This adds some latency to a user interface that cannot be eliminated due to limitations imposed by physical laws.

- There is some extra load on the application server that can result in additional resource consumption.

Unfortunately, without a supplementary library Ajax does not fully solve the problems related to accidental complexity and lack of expressiveness of imperative validation code. As well, it does not mitigate common security issues. However, XMLHttpRequest is widely supported even on mobile platforms [76] and certainly can be used as underlying communication protocol for both in-line and post-submit validation.

4.10 Web application firewall

Most of the methods described in this chapter require knowledge about application data flow. But in absence of this knowledge you can still apply some trivial data sanitization techniques with Web application firewall (WAF). WAF is a software system or a hardware appliance that sits between a client and web server and which aim is to shield the server by determining patterns that violate the security policy in incoming HTTP(S) request/response packets [14]. If the packet is considered safe, then it is sent to the main server; otherwise, the client connection is closed. The security policy is usually applied with accordance to the WAF configuration file and built-in checks. Most of web application firewalls promise protection from such attacks as XSS, SQL injection, and detection of abnormal behavior that does not fit the website's normal traffic patterns. A widely deployed example of WAF is an open source module to an apache web server — Mod Security.

Although web application firewalls can be used to guard backend server, their scope of applicability is limited, mainly because of the low level of integration with an application itself. This factor makes WAF advisable but not sufficient for comprehensive website protection.

4.11 Summary

Most input preprocessing techniques for online applications were developed at the beginning of this century, and in spite of their rapid advancement, there are still some issues that validation frameworks have in common. In our opinion, one of the root problems is their focus on initial experience of use of constructs rather than security and code maintenance characteristics. While adding a validation rule right to a view and then binding it to a model or business layer is not difficult to implement, in the long term perspective, this approach does not scale well. In such a case, changes in requirements demand modification on several layers, and since request validation is usually splashed all over the codebase, in order to grasp how a certain parameter is validated one would need to construct the whole picture from different places. This introduces notable accidental complexity and is not acceptable from the code support point of view.

All of the existing solutions that were studied include some set of validation utility classes that can be utilized for data preprocessing. However, most of them provide only basic validation/normalization rules and generally do not include a toolkit that is sufficient enough for rapid application development. Due to lack of uniformity, developers often need to write boilerplate code and validators which might not take into account all traits or have security holes.

Most validation frameworks focus only on processing of web form input and do not support the same interface for a URL path, regular GET/POST parameters, cookies, and headers which can be also part of the input. Finally, we were not able to find a solution that, besides its main validation capabilities, facilitates best practices in usability of data input. As they are quite crucial for providing a good user experience, ignoring them can result in a direct revenue loss.

Chapter 5

Framework Requirements

Detailed analysis of existing methods and techniques for server and client side validation allowed us to define requirements that will reflect the most desired features of a rule engine for input preprocessing in web applications, and, consequently, we defined the following objectives for our research:

- Provide a methodology that decreases accidental complexity of validation code, lessens the effort required to maintain it, and suggests a data preprocessing mechanism that is transparent for rule developers as well as external auditors (e.g. security analysts).

- Develop a framework for validating external data both on the client and server sides. It should cover the whole spectrum of validation stages and support elaborate rules scenarios without sacrificing much of the code's readability.

- Suggest a solution that can improve some aspects of website security without requiring a solid background in input validation. The framework should also generate data necessary for monitoring suspicious behavior and intrusion detection.

- Design an API that allows a client user interface to cleanly integrate with the rules engine and customize feedback for both inline and on-submit validation.

- Support 90 percent of the typical input validation tasks out of the box, and for those validators and converter not included, provide a flexible interface so that any task not covered by the library will be easy to complete.

We firmly believe that fulfilment of these objectives, along with taking into account problems in existing schemes, can result in a solution that will radically change the

current approaches to input validation. In the following sections we will supplement our statements and explain the details.

5.1 Centralized input preprocessing

The security problems described earlier in this document, in particular possible slipping of raw data into an application model or view, lend credence to the theory that there should be a separate layer between a client and page controllers that encapsulates all required validation and sanitization logic. In this way we can ensure that whenever a trust boundary is crossed the data is always clean and secure to use.

One of the methods of isolating a program from unprocessed input is a barricade pattern described in [5]. Generally, it has the same concept as firewalls in buildings and designates certain interfaces as boundaries to "safe" areas:

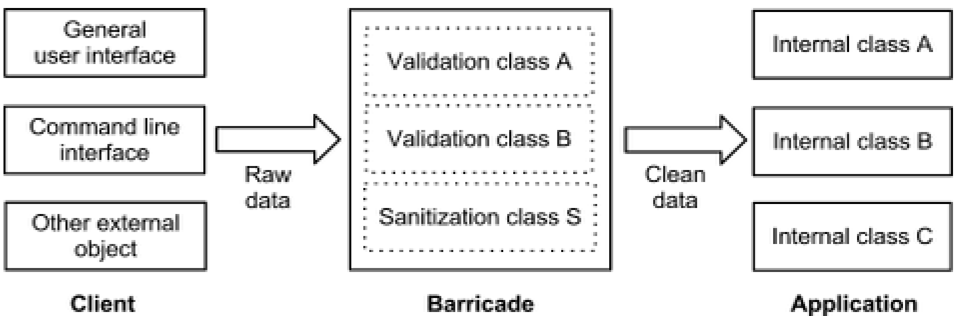

Figure 2: Barricade pattern

The pattern ensures that no external value can get to the application’s internal classes and bypass preprocessing by the barricade. This includes GET and POST HTTP parameters, cookies, values embedded in a URL path, and optionally HTTP headers such as a user agent string and an IP address. Hence, there will be no slips of the raw values to view or model.

The barricade is a good example of the separation of concerns — the process of separating a program into distinct features that overlap in functionality as little as possible [37] — and we expect that applying this practise to our case by isolating a validation system to an individual tier can notably simplify the design of client-server applications [72].

First of all, such modularity mitigates security issues described in Chapter 4 by isolating the business logic from suspicious entities that are possibly malicious and making a developer explicitly define boundaries for each input value rather than considering validation as an optional artifact. Secondly, a separate layer evolves the system into more maintainable as by superseding validation scattered haphazardly throughout multiple application layers with a common business and security policy stored in one place, we make input preprocessing more transparent to the developer. In addition, by abstracting out the validation system from the markup and model we keep them readable and not cluttered with business logic, and at the same time permit to unit-test the rules separately from controllers and views. This abstraction also allows us to have separate barricades between different layers, so, for example, on POST request we can apply format and business rules but escape HTML characters only when data reaches the presentation layer.

Another important advantage of this pattern is that once preprocessing rules are separated from other logic, they can be expressed in a way that is independent from the platforms used by the server and client sides. For instance, one can use a domain-specific language (DSL) oriented on data preprocessing which would make the code easier to comprehend and maintain.

On top of that, such a design has many practical benefits. Since the source of input should not make any difference for the barricade, the same preprocessing system instance can be used for more than one client or server application, so the rules can be applied in each place where the target input is used without having to duplicate any logic. The intermediate tier allows to cut off malicious data before it reaches the controller, which is crucial when a server is under a DoS attack that operates at the OSI application layer, as in this case it can be repulsed more effectively. Owing to this isolation, applications can also take advantage of service-oriented architecture to ensure near linear system scalability.

While the barricade pattern is a robust way to filter untrusted input, some clarification to the original McConnell design is necessary to make it flexible enough for online applications. The original design assumes only a one-way interaction between a client, the barrier and a main application, however, in some cases the validation system should have access to a current model state or the ability to control a page view in order to evaluate part of the rules directly on the client. Thus we need to define a two-way interface between the barricade, business and presentation layers to make such interaction feasible. Luckily, it is easy to achieve with callbacks and asynchronous HTTP requests.

5.2 Whitelist validation

To reduce potential security risks all external values must be checked against some data type and/or string format validator, so no HTTP parameter without a corresponding validation rule can pass through the barrier. Furthermore, each value should be sanitized by a set of default converters. All exceptions to these requirements must be explicitly defined on a configuration or rule level. In this way we make sure that the users will not be able to intentionally or inadvertently bypass the preprocessing provided by the framework and the application will always operate on clean data.

We expect that these requirements, along with powerful utilities for validation and sanitization of text data, will create an environment that will help even an inexperienced developer to design applications that do not suffer from vulnerabilities related to incomplete input preprocessing.

5.3 Cross-site request forgery and protection from bots

One of the ways to discriminate human activities from computer-based actions and prevent a cross-site request forgery attack is to ensure that the client can execute the JavaScript code and use a system based on securely generated tokens. Similarly to what is proposed in [47], our approach can be defined as code execution with the purpose of validation, but in addition to protection against web robots we also defend an application from CSRF by generating a token on a server instead of a client. More specifically, the form submission process is based on a two-phase functioning:

1) At first, the client requests a web page and the server provides a response with a secure token. A token generation function might calculate hash from some random value combined with a "salt", a session cookie and an IP address from which the user requested the page.

2) When the data is ready for submission, the client library composes a post request with the obtained token which is then validated on the server side.

The only way to correctly submit a form is to send an XMLHttpRequest with a validation token, while regular submission without a token will result in a decline of the request prior to any other operation. Therefore, a web robot without support for a client side language can be effectively stopped by this strategy (unless it was specially programmed for the target application, in which case there is no guaranteed

protection [119]).

As a drawback, the described protection will make the site inconvenient for users with turned off or inaccessible JavaScript. Though, with extensive improvements of the script language support in modern web browsers (including mobile ones [50]) it should not be an issue for the vast majority of users. Ultimately, the framework can allow to disable tokens from the configuration file.

5.4 Expressive language syntax

When it comes to accidental complexity, we strongly believe that one of the main goals of programming language and framework designers is to eliminate as much of it as possible. At the same time, from the perspective of the users of these tools, choosing one that is particularly designed for solving problems from the given domain should help to considerably lower application complexity.

One of the benefits of the barricade pattern is the possibility to use a separate domain-specific language for expressing input preprocessing rules. By providing notations tailored to the domain under consideration we will be able to share a common metaphor of understanding between software developers, security auditors and business analysts, and thus allow them to reason about the rules as encoded in an application's infrastructure and verify that they implement all documented requirements. A DSL that can express the model in a simple and concise way, i.e. one that is not cluttered with boilerplate code, offers a substantial gain in development productivity and, if well-designed, induces less error-prone code in comparison to general programming language counterparts [54]. One can argue that use of this approach incurs the cost of familiarization with new semantics of the language, but from our experience these efforts are actually comparable to learning an analogous framework API.

Declarative DSLs have acquired the reputation of a reasonable choice for rule engines [30] and generally are the most suitable for static code analysis. Furthermore, mix of declarative and functional styles makes it possible to write code in a way that more closely resembles human reasoning and thus is easy to read and understand. For example, the validation rule expressed in declarative way:

```
id[0] stripTags >> toPositiveInt [m.notInt] >> unique [idNotUnique]
```

has much less syntactic overhead than the same logic implemented in an imperative language:

```
String id = stripTags(map.get("id"));
if (id.isEmpty()) {
  id = "0";
}
if (!isPositiveInt(id)) {
  throw new ValidationException(m.getString("notInt"));
}
int intId = Integer.parseInt(id);
if (!unique(intId))
  throw new ValidationException(idNotUnique);
}
```

Hence, we deem that integration of a DSL that supports both functional and declarative paradigms can be part of a solution of the accidental complexity problem.

Using our domain-specific language, a developer should be able to designate validation and canonicalization requirements by defining the corresponding preprocessing rules, and then at runtime, all external data must checked against these rules under the system control.

5.5 Client side

One of the goals of this research is to develop a framework that allows a software engineer with minor experience in designing interfaces to get some improvements on site forms. The client library should take into account the best practices in user experience established during the last decade and make their usage as easy as possible. The ultimate goal is to move user data input from cognitive to fluent type of user-product interaction [83].

A good example of a UX practice that simplifies filling in of web forms is a validation technique that provides confirmation whether a suitable value was provided at a moment when a user enters data within input fields and prompts to make changes prior to the form being submitted [17]. This technique is called in-line validation and makes correction of ill-formed entries much more easier — for instance, Luke Wroblewski [10] claims that this technique accelerates form filling by two times. According to the results of his research, in comparison to a page submit/refresh model, it gives:

- 22% increase in completions
- 31% increase in satisfaction ratings
- 42% decrease in completion times

- 22% decrease in errors made
- 47% decrease in number of eye fixations

Besides inline validation, the client library should use flexible mechanisms for customization of error events triggered on the client side. There are also some minor features which should be supported to make a user interface more friendly: feedback on successful validation that increases the user's sense of accomplishment [83] (like a green tick that appears next to a field), disabling the post button to avoid multiple submissions, and setting the right order of tab navigation [120]. The same preprocessing server should be able to operate with several types of clients: a browser, an iOS application, and a java application.

5.6 Library of common converters and validators

According to Microsoft Research [48], 90 percent of validation tasks on websites are common operations. A provided library of converters and validators should cover all of these cases and allow to apply rules defined in a DSL script without a developer having to write a validation code for the client side.

To make an interface loyal to application users, the system should not impose a stringent format for the input as a user usually just wants to get something done, not think about the correct way to type the data. An interface design pattern that permits a user to enter a value in a variety of syntaxes and makes an application interpret input data is called forgiving format. In support of this practice, intelligent converters for the most common data types must be implemented in the library, so that, for instance, a user should not need to guess a format of a zip code or telephone number.

5.7 Summary

To recap the functional and nonfunctional requirements that were stated in this and previous chapters, we expect the following features to be present in the validation system:

- Preprocessing rules syntax and semantics should be highly expressive.
- Input preprocessing must be extracted to a separate application layer.
- Unsafe patterns must be escaped unless they are explicitly allowed.

- The rules must be applied automatically on the client side.
- There must be in-built protection from CSRF and web bots.
- The system must include a library with common validators and converters.
- The solution must be developed for a relatively popular web platform.
- In-line validation and forgiving format must be used whenever possible.

Chapter 6

Input Data Flow

Considering the previously defined requirements, in this chapter we suggest a process that manages request preprocessing for web applications, starting from generation of a form to creation of submission confirmation page. We will describe the sequence of events that should occur before a request can be processed by a business layer both on the client and server sides. The process is described for a web site, but a similar design can be applied to any client-server application.

6.1 Form validation

The web form validation process consists of the following steps:

1. A client sends an HTTP request with a page URL to a server.
2. The web application generates a cryptographic token for each form on the requested page that uses the POST method. Then the tokens are stored in a memory location that can be read by any preprocessing server and a resultant HTML page with the forms and a library for client side validation is returned to the user's browser. The tokens can be created using a secure random generator and a hash function with salt [106] (e.g. SHA-256 that depends on username and client IP).

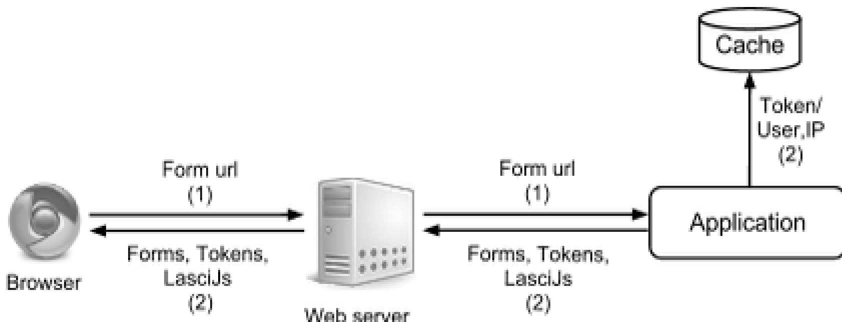

Figure 3: Form validation, steps 1-2

3. When a page is rendered, the initialization phase begins. During this stage an XMLHttpRequest with a controller path URL for each form is sent to the server (the path is taken from an `action` attribute).

4. The server preprocessor finds the custom client rule for each form element (see Section 11.3) and returns a result as a JavaScript file. After the response is received, the client library instruments the HTML document with callbacks which apply the returned validation rules upon user-input events.

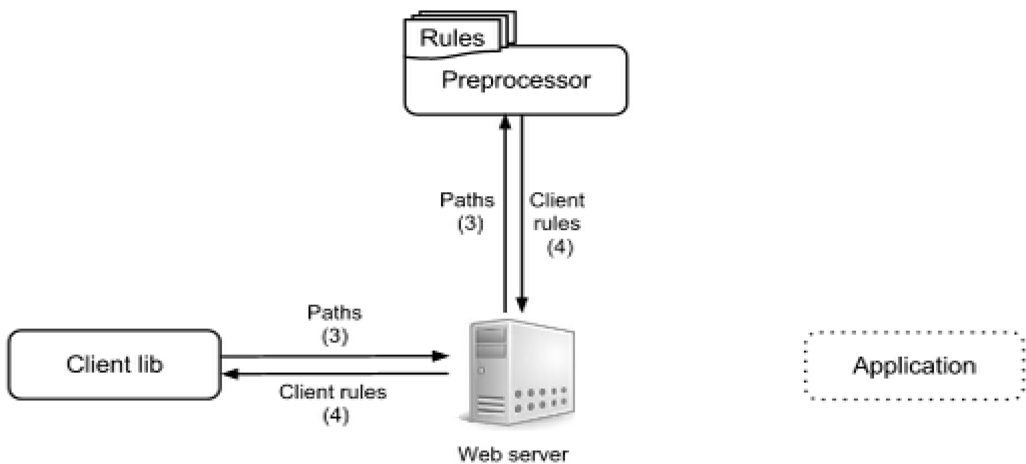

Figure 4: Form validation, steps 3-4

5. The user enters data, and when some value needs to be validated (by default when an element loses its focus), the library looks for the corresponding validation handler on the client side and executes it. If the rule fails at this stage an error message is shown to the user.

6. If a rule cannot be processed on the client, an asynchronous HTTP request to the server is made. The request contains the parameter name, all known

input values, and a form controller URL.

Figure 5: Form validation, steps 5-6

7. On the server, at first, the preprocessor evaluates all rules that precede the rule for the currently validated parameter, and then applies the rule defined for the parameter under validation (the first step is needed to ensure that all parameter dependencies will be satisfied). If validation depends on data from some third-party service, like CAPTCHA, the rule engine makes an external call.

8. The result of validation is sent back to the client and if the value appears to be invalid:

    a. A focus is set to an element with an invalid value and a client *show error* handler is called, following which a user should type another value that matches the criteria.

    b. When the value needs to be validated again, the rule is reapplied and the error message is hidden using a client *clear error* handler. The message is shown again if the new value does not pass validation as well.

The error message is not shown if the user changed the input value after the asynchronous request was made or if he changed a value a rule for which is defined before the rule that is currently applied. Instead, the client waits for the most recent request response to provide up-to-date feedback on entered data. Also at this stage, the client library can redirect the user to an error page or perform other actions based on the server response.

When the value is proven to be valid, the validation request is sent for all non-empty inputs, the rules for which are defined after the last applied rule.

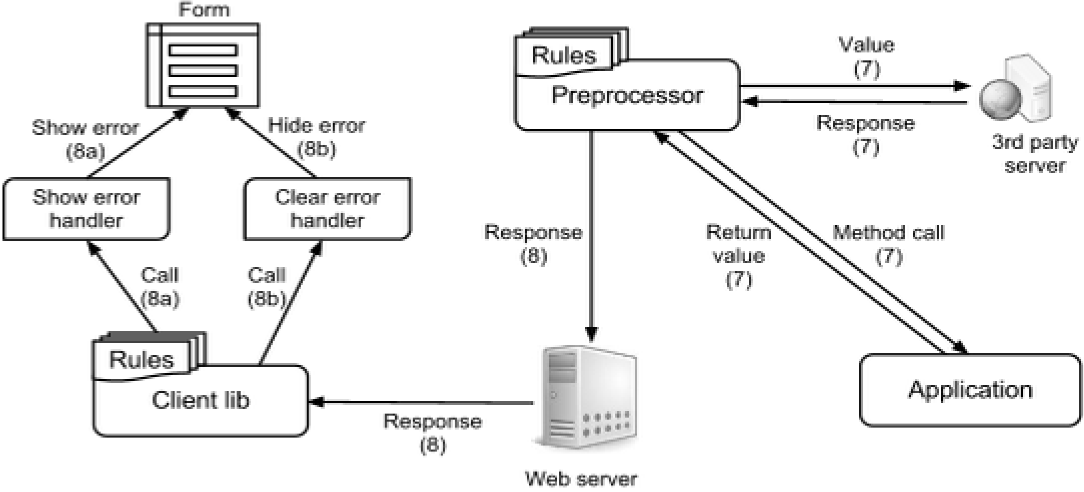

Figure 6: Form validation, steps 7-8

9. On a submit event, form values (as were typed by a user) and the form token are sent to the server via XMLHttpRequest.

10. The server preprocessor calls an application callback to check the token validity. If the token is invalid an error message is shown to the user.

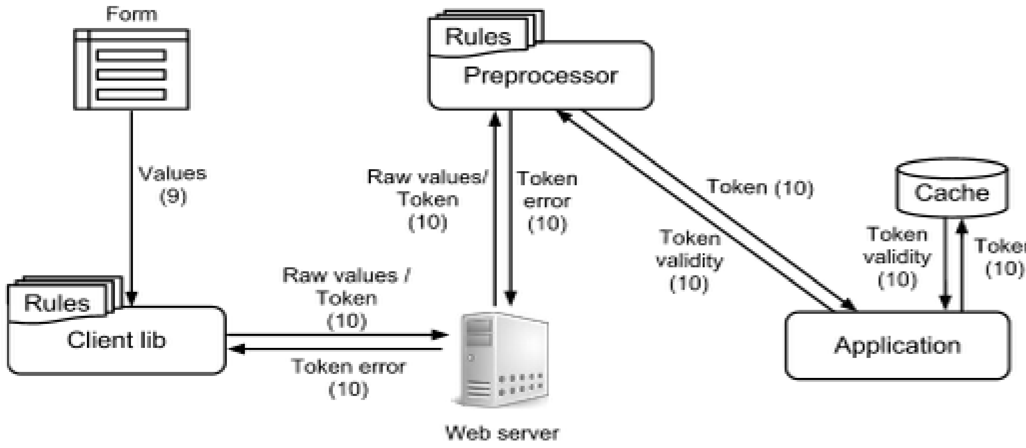

Figure 7: Form validation, steps 9-10

11. If the token was valid, the preprocessor reapplies all the rules to ascertain whether an along-the-wire tampering has taken place, then:

    a. In case of any errors during preprocessing of submission data, the server responds with an error report.

    b. If all inputs have passed validation, a request wrapper substitutes original parameter values with ones that were validated and normalized to a form acceptable by the main application. Then the request is passed to a controller, which processes the data and creates an HTTP response with a redirect URL, a custom JSON response, or a list of error messages.

12. In case of a negative server response, in addition to the actions from the 8th step, a custom client *submit error* handler is called; otherwise a *success submit* handler is called and the page is redirected or substituted with new content (see Section 11.4 for details).

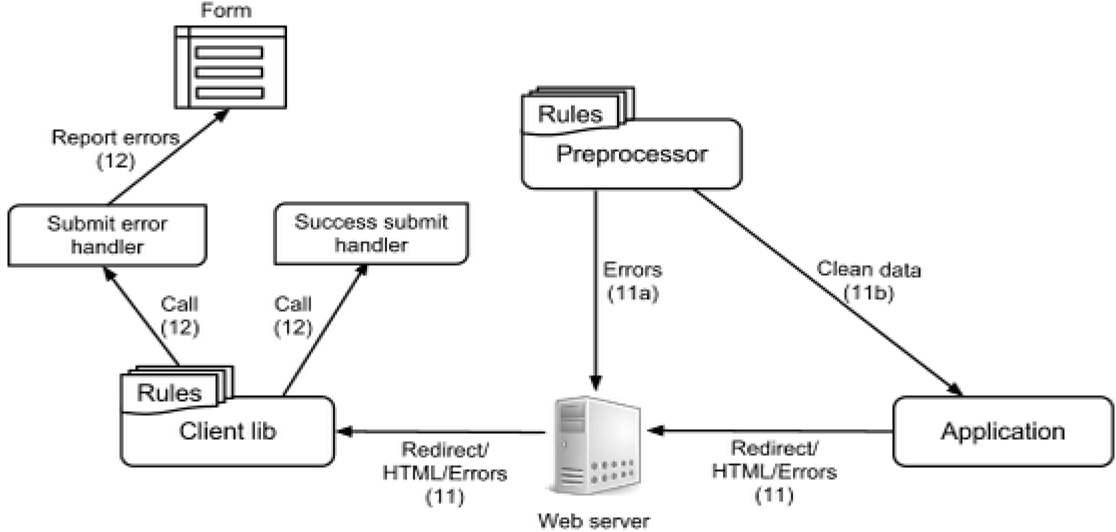

Figure 8: Form validation, steps 11-12

6.2 Validation of navigation request

Each request should be processed by a validation filter before it reaches a controller. This allows to efficiently validate not only form input but any GET or POST request. Assume, for example, that there is an online application where a user creates a profile and then others can access it under the following URL:

```
http://example.com?user=<id>
```

To make sure that the `id` value is a natural number and corresponds to an existing profile, type check and business-logic rules can be applied.

The general algorithm for get parameters and cookies is pretty simple and consists from the following three steps:

1. A client sends an HTTP request to a server with a page URL.

2. The preprocessor applies a validation rules set that matches the target controller:

    a. If any parameters are invalid, the browser is redirected to a page with an error description.

    b. If the request is valid, normalized data is passed on to an application.

3. The application responds with page content.

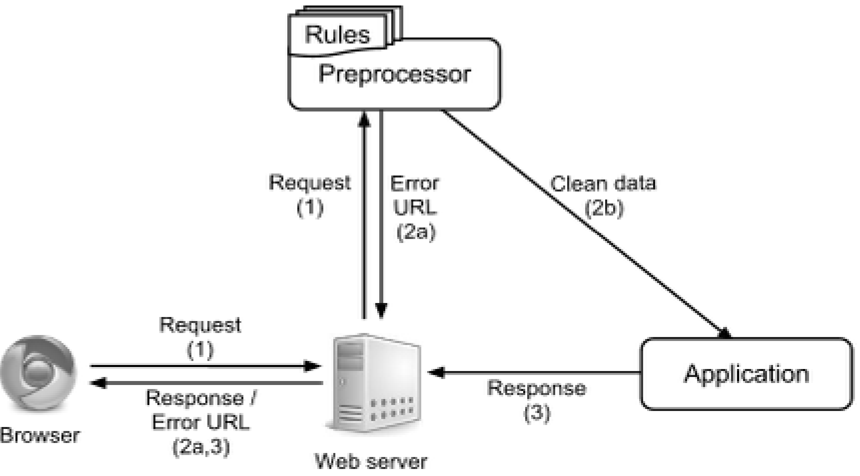

Figure 9: Validation of a GET request

6.3 Summary

We have found the depicted infrastructure to be a suitable solution for the data input lifecycle as it consistently ensures model integrity and security by storing all

preprocessing rules in one place, and at the same time, it stays resilient in terms of available customizations and requires a minimum amount of boilerplate code. By leveraging inline validation via Ajax and custom code on the client side, we optimize user experience and ensure that application performance stays high. At the same time, the security tokens increase protection from cross-site request forgery and web bots.

Chapter 7

Choice of Groovy DSL

In this chapter we compare internal and external DSLs and justify our choice of Groovy DSL as a solution that is most capable of meeting our needs.

7.1 Pros and cons of Groovy DSL

One of the challenges we faced during this research was a choice between external and internal DSL. External DSL is a language that is implemented as a compiler, translator or interpreter, while an internal one is, essentially, embedded into some pre-existing general purpose programming language (a host language) using its subset as the custom syntax for a new abstract language.

While you can define any syntax and semantics for an external DSL, such languages have one essential disadvantage — lack of symbolic integration, i.e. that they are not linked into a base language [56]. Take for instance calling a method implemented in a main application language from a DSL or vise versa. In the case of external DSL you need to implement a special interface between two methods, while with internal DSL we get this interface for free. Other examples would be debugging, autocompletion, and renaming of the class properties — with a modern IDE they are habitual, but in case of external DSLs, support of these features require installation of additional tools that understand the semantics of your language.

After careful consideration of most suitable options, we chose internal Groovy DSL as a front-end of our rule engine, primarily because Groovy:

- Compiles to JVM bytecode and is compatible with Java which allows to call main application methods directly from the scripts or vise versa. This ensures seamless integration of the DSL with an existing Java project and makes it a suitable choice for many potential deployment servers.

- Allows to write a DSL with minimally intrusive and readable syntax, e.g.:

```
// paint(wall).with(red, green).and(yellow)
paint wall with red, green and yellow

// check(that: sky).is(blue)
check that: sky is blue
```

- Aggressively evolves as a host language for internal DSLs [9], and already supports most desirable features: flexible syntax, metaprogramming, compile-time transformations, dynamic types, closures etc.

- Has DSL descriptors [79] that significantly improve content assist in IDEs.

- Permits to limit available types of expressions in the designed DSL and in this way reduce false-positive compilations.

- Uses `BigDecimal` instead of doubles for numeric literals [80] which in our case simplifies handling of decimal values like price, size, and weight.

- Supports static typing and type inference for individual methods and classes [109]. By using inference-based typing we do not need to sacrifice the static check available in Java and thus can ensure type safety of validators and converters. At the same time, we can leverage Groovy dynamic nature to define an expressive syntax for the preprocessing rules.

- Provides flexible runtime code evaluation via metaprogramming facilities.

- Gives access to abstract syntax tree (AST) during a compilation phase. It allows to compile scripts to Java bytecode in advance and automatically emit new client code each time a new rule is added or an existing one modified.

- Supports scripts and has a script engine that complies to JSR-223 [8]. This eliminates the need to install a Groovy runtime environment.

- Has an increasingly popular web framework called Grails (according to Google Trends [122]). Integration of the DSL in this framework will make it accessible for many web developers.

However, there are still some disadvantages of using internal Groovy DSL:

- Internal DSLs are limited by the syntax and structure of a host language.

- While certain types of expressions can be prohibited by a language designer, because of non-exhaustiveness of this method a user is still able to write expressions that go outside of the DSL grammar boundaries.
- Error messages can be unsatisfactory for understanding the original source of the error (it is a common problem of internal DSLs [55]).
- The Groovy metaprogramming model requires dynamic method dispatching which is relatively slower than Java `invokevirtual` instruction [61]. However, recent benchmarks show notable improvements in this area [69], so we expect that performance will not become a stumbling block for our users.

With internal DSL we have gained all the benefits and drawbacks Groovy affords us, and after examining them we decided to proceed with this choice as, in our opinion, the capabilities of the language mitigate most of the existing problems. Other factors that supported our decision were successful integration of Groovy as a language for a rule engine in Oracle Fusion Middleware [7] and its employment for expressing pre- and post-conditions in OVal [13], a popular object validation framework in Java.

7.2 Other considered options

The main competitors of Groovy in the DSL for JVM sector are Scala and Clojure. We have studied both of these options and found that, in terms of our problem, we would not be able to implement a language with such an expressive syntax. In the case of Scala, its static typing limits us in terms of dynamic features, while Clojure's Lisp syntax is not compatible with Java, which is inconvenient for writing inline closures.

We also had a look at the third kind of DSLs, which are language workbenches. The main reason why we have not chosen them are additional tools that need to be installed to leverage their functionality. This makes language workbenches an option only for languages aimed at narrow domains.

Chapter 8

DSL Syntax and Semantics

Considering the requirements in Chapters 5 and 6, we have come up with a DSL called Grules [108] that is simple, expressive, and flexible enough to support the necessary features. We have analyzed many existing frameworks and visited a large number of web pages to determine the sort of scenarios preprocessing rules needed to be able to handle. In this chapter we define fundamental terminology, introduce the proposed language grammar, and describe the types of rules that can be declared in grules DSL.

8.1 Rules and functions scripts

There are two kinds of files that can be defined in a grules DSL:

Rules script — a script that contains specification of preprocessing rules that must be applied to incoming data. The main purpose of a rules script is to ensure that all external values have been properly preformatted and sanitized before updating an application model. Subsequently, after the input is processed by a rules script it may be considered clean.

Functions script — a class that defines validation and conversion functions used by the rules scripts.

To be evaluated as a rules script, the file name must end with the suffix `Grules`. As a functions script you can specify any Java or Groovy class with static methods. For convenience, you can use the class-level `@Functions` annotation, which adds the static modifier to all class methods at compile time, for example:

```
@Functions
class StringFunctions {
  String trim(String value) {
```

```
    value.trim()
  }

  String contains(String value, String text) {
    value.contains(text)
  }
}
```

8.2 Validators

A validator is a function that examines its input with respect to some requirements and produces a boolean result which indicates whether the input successfully passed the validation. This snippet shows how to implement a validator that tests if the supplied value is even:

```
boolean even(long value) {
  value % 2 == 0
}
```

As grules is the internal Groovy DSL, the return keyword is optional at the end of a function body. The return type of a validator should be `boolean` or `Boolean`. The other option to indicate a failed validation result is using a `ValidationException` (see Section 10.3).

The first validator's parameter is called a processed value parameter, as it represents a value of a parameter processed by a current rule (you will learn more about rules in the next chapter).

8.3 Converters

Converter is a function that sanitizes a value of a parameter according to the context in which it will be used, reduces the value to a canonical form acceptable by a main application, or applies some other custom transformations. The simplest example would be a `trim` function that removes spaces before and after the value:

```
String trim(String value) {
   value.trim()
}
```

Another example would be a converter for a postal code that, for example, transforms k1A 0j9 into K1A0J9.

A converter can also take regular parameters:

```
String toDate(String date, String format) {
   //...
}
```

It can return a value of any type including dynamic (`def`). Both converters and validators can be overloaded or overridden.

8.4 Rule application

A preprocessing rule defines a sequence of converters and validators that must be applied to an input value before it can be processed by a controller.

The syntax for rule application declaration is:

```
parameterName rule
```

where `parameterName` is a valid Groovy variable name that starts from a lowercase letter.

For example, this grules script normalizes `categoryId` and `login` parameters:

```
categoryId trim //categoryId - parameter name, trim - converter
login isAlnum
```

If a function (validator or converter) is of arity one, its single argument is implicitly equal to the value of a parameter processed by a current rule or a value passed from the previous converter if such exists, so `categoryId trim` actually means: call the function `trim` with a `categoryId` entry content and supersede the original value with a trimmed one. Analogously to the first rule, the `isAlnum` validator is called with the string value of the `login` parameter. The only difference is that if the function returns `false`, a validation error occurs.

To use a function with more than one parameter, just omit the first argument in the function call. For example:

```
id add(id, 10) // add value 10 to id
```

8.5 No-operation converter

If some value must be passed to the program in its "raw" form without any conversions or validations, one can use the `nop` (no operation) converter:

```
message nop
```

In this case, grules will skip validation for the `message` parameter and leave the value unchanged (checks for non-blank entries and default converters/validators are still applied).

8.6 Composite rules

An input value frequently requires the application of more than one validator or converter. For example, you might want to check whether or not a string in a text box is a valid date, and if it is, whether it comes before some deadline. Composite rules provide a simple function chaining mechanism by which multiple functions may be applied to a single datum in a defined order. If an input value does not meet requirements, the system provides information about which part of the rule it did not pass. For example, if a chosen date does not satisfy one or both of the requirements, the system will let the client know which of them it fails to meet.

Validators can be combined via conjunction, disjunction and negation operators (`&&`, `||`, `!`). Rule evaluation is done in compliance with the precedence of logical operators in propositional logic — conjunction has higher precedence than disjunction. Also, similarly to Java logical AND and OR operators, grules uses short-circuit evaluation [6]. Further, we will call a rule built only from validators and logical operators a *validation rule*.

The chain operator (`>>`) is similar to a Unix pipeline [65] and divides a rule into independent parts, where each subrule can be either a single converter or a validation rule. Subrules are applied from left to right and if one of them fails, the rest are not checked.

Let's look at some examples. In each case we will explain how a parameter value is normalized and validated by the rule engine:

1. `id trim >> stripTags`

This rule is defined as composition of two subrules and says that a value of the `id` parameter at first must be normalized by the `trim` function and then the result must be passed to the second subrule, the `stripTags` converter. After that the main application is guaranteed to use a clean value of the `id` parameter.

2. `login trim >> !isAdmin >> stripTags`

   Here we have three subrules: `trim`, `!isAdmin`, and `stripTags`, so the result of `trim` application is checked against the `!isAdmin` validator upon which the value is passed to the last `stripTags` converter.

3. `login isAlnum && (isLengthMore(6) || eq('admin')) >> toLowerCase`

   There are two subrules in this rule: `isAlnum && (isLengthMore(6) || eq('admin'))` and `toLowerCase`. Here, the parameter value is checked against the `isAlnum` validator and the `isLengthMore(5) || eq('admin')` validation rule. If both validation rules succeed, the value is transformed to lowercase by the last converter.

Such a rule syntax eliminates syntactic noise that is present in many imperative programming languages and uses a straight sequence for function applications.

One may note analogy with monads [87] in our rule semantics, and indeed the chaining (>>) is a bind operation which takes a parameter value and function that encapsulates preprocessing logic and passes the result of the validator or converter application to the next function or constructs a validation error. We can prove that subrules satisfy three monad axioms:

- Right unit:

  `f >> {return it} ≡ f`

- Left unit:

  `{return it} >> f ≡ {f it}`

- Associative:

  `m >> f >> g ≡ m >> {g(f it)}`

8.7 Optional parameters

By default a rule implies that an input parameter is required and can not be blank or absent. To define an optional parameter you can add a default value in square brackets to the start of a rule, for example:

```
id[defaultValue] toPositiveInt
```

Where a `defaultValue` is any Groovy expression that evaluates to a non-void type (can be an empty string). Now, each time a client leaves the parameter blank its value will be substituted with `defaultValue`. Please note that default values are evaluated on each script run which may be relevant for a variable that relies on an object like an instance of `java.util.Date`. For example, one can easily use a current date to fill an optional `transactionDate` field:

```
transactionDate[dateFormat.format(new Date())] toDate('yyyy-MM-dd')
```

Here, the `Date` object is created each time the parameter `transactionDate` is validated.

8.8 Type conversion

The logic for conversion of a parameter value from `String` to one of the other supported types is defined by type converters: `toInt`, `toBoolean`, `toBigDecimal`, `toChar`, `toLong`, and `to<Type>List`, where `<Type>` is `Int`, `Boolean`, `BigDecimal`, `Char`, or `Long` (you can find more about type converters in Section 13.2). The job of these converters is to check that a value of a parameter can be transformed to a target type and then return a result of the conversion. A converter throws the `ValidationException` if the value can not be casted to the corresponding type. As an example, to validate an integer parameter that has to be greater than ten you can use the combination of functions `toInt` and `gt` (greater than):

```
id toInt >> gt(10)
```

8.9 Accessing other parameters values

All rules that we went over above accessed only one parameter. However, to test whether an input satisfies all preconditions, sometimes you need a rule that is defined as coordination constraints between two or more parameters and hence requires access to multiple entries at once. A simple example would be the requirement that in a registration form an original password must match the value supplied in the confirm field:

```
passwordConfirmation eq(password)
```

Another example is a rule that compares travel start and end dates:

```
arrivalDate isBefore(departureDate)
```

A multi-input rule can refer to a parameter already checked and normalized by a previously applied rule (clean data), as well as to a parameter value before any preprocessing (raw data). A clean value can be accessed by just a parameter name, while a raw value should be referred as `$parameterName`. If a clean value is not available because the accessed parameter failed validation, any access to it will cause the current rule to be skipped. If a corresponding rule has not been applied yet, the engine throws the runtime exception `InvalidParameterNameException` (this exception is ignored for inline validation).

8.10 Rule groups

Grules supports groups that can represent different types of parameters. Each type is defined in a separate section marked by the appropriate label. For example, for web applications grules predefines six groups: `GET`, `POST`, `COOKIES`, `HEADER`, `URL_PATH`, and a `PARAMETERS` group, which combines both GET and POST parameters (the last group should be used only when a list of allowed request methods is explicitly set, see a Grails controller class for an example).

Here is a rules script for a login form that uses groups:

```
GET:
id toNaturalInt

POST:
```

```
username isAlnum
password isPasswordFor(username)

COOKIES:
sessionid isSessionIdFor(username)

HEADER:
referer isUrl
```

Most likely, you will only need the `HEADER` section only on occasion, but if you do use any header parameters, make sure they are validated properly as these values can be easily modified by a malicious user. For example, an attacker can modify the referer HTTP header field which contains the URL of the web page from which the request originated, or set the `Content-length` header to some unduly big value [32, 33] (you will read more about the `URL_PATH` parameters in Chapter 9).

One can have more than one section of each type to satisfy cyclic dependencies between groups, for example:

```
POST:
user isAlnum

GET:
page !empty(POST.user) && (eq(PageNames.HOME) || eq(PageNames.ADMIN))

POST:
if (GET.page == PageNames.ADMIN) password isPasswordFor(username)
```

By default, the rule engine looks for the referenced parameter only in the current section. To access other parameters use the section name prefix followed by a dot, for example:

```
COOKIES:
lastUsername eq(GET.username) // GET.$username for a raw value
```

A developer can also define custom sections:

```
TOUR_INFO:
country isCountry >> toISOCountry
city isAlnum
...

PERSONAL_DATA:
name isAlnum
surname isAlnum
```

8.11 Closures

When a function is not expected to be used more than once and is simple enough to fit in one statement, you can define such a validator or a converter as a closure, for example:

```
guestsNum toInt >> {it > 1}
```

The `it` variable inside the closure refers to a value passed from a previous subrule (here `toInt`). Also, since closures are part of the rules you have an access to all input parameters, e.g.:

```
adminsNum toInt
guestsNum toInt >> {guestsNum + adminsNum < maxUsersNum}
```

You can also use a closure validator or converter to explicitly define all function parameters, for example, to run a function against a parameter that is not the default for a current rule.

Validator and converter calls that occur in a closure body are handled as general Groovy method calls and thus all function arguments must be specified. For example:

```
// expectedChildrenNum is a method that takes one argument
expectedGuestsNum toInt >> {it + expectedChildrenNum(it)}
```

Using a closure you can easily assign a constant value to a parameter, for example:

```
tempParameter {7}
```

8.12 Combined parameters

It is often reasonable to process several input values, like radio buttons or checkboxes, as a group. To define such behavior you can wrap parameters in a list and specify it as the rule's target value. For example, the following line checks that both `accept` and `subscribe` check boxes are selected:

```
[accept[‘’], subscribe[‘’]] toBooleanList >> isEvery
```

8.13 List parameters

The rule engine supports the use of list parameters, which usually represent a check box group or a list of inputs with the same name. For example, in HTML you can specify the latter using the `name[]` expression:

```
<input type="text" name="options[]" value=""/>
```

The syntax for list parameters does not differ from other parameters, so for instance, to set a validation constraint for the above field you can use the rule like this:

```
options isEvery(eq('confirm'))
```

In this case the `eq` function is applied to each element of the `options` list, and the rule fails if at least one member of the list does not pass validation (see also `isAny` and `collect` functions).

8.14 Custom parameters

While browsing the Internet you may find that it is common for a web form to contain a value that is based on entries from multiple fields, such as a birth date or a full name. To define such an assembled parameter one can use the `@Parameter` annotation, e.g.:

```
@Parameter
fullName = name + ' ' + surname

fullName isAlphaSpace
```

Here, the parameter `fullName` is constructed from two other parameters (a name and a surname) and validated by the `isAlphaSpace` validator, then an assembled parameter is available to a main application along with other parameters.

8.15 Conditional rules

In case your validation checks depend on values of other parameters or environment variables, it might be required to trigger a validator or a converter only if a certain

condition is met. To implement such behavior one can use a Groovy ternary operator enclosed in parentheses:

```
// createCheckBox is a boolean parameter name
username (createCheckBox ? isUniqueUsername >> ... : nop) >> ...
```

For more complex conditions you can use an IF statement. For instance, to process a `username` and `email` field only if a `createCheckBox` is selected and a signature field is non-blank the following snippet can be used:

```
if (createCheckBox && !empty($signature)) {
  username isUniqueUserName
  email isEmail
} else {
  ...
}
```

The predicate expression can contain any expression that is valid in terms of Groovy semantics.

8.16 Default converters and skip function

In an effort to improve code usability and security, some preprocessing functions like `trim` are applied by default to all input values. Thus, a developer is able to implement whitelist validation (see Section 5.2) by explicitly marking parameters for which security checks can be deactivated or apply certain canonicalization converters to all input values.

To adjust the preprocessing policy you can either change a grules configuration file or apply the `skip` utility function. This function can be used on a per rule basis and indicates which default converters and validators have to be skipped for a certain parameter. As an example, the following rule says that the `trim` function need not be applied for the parameter `message`:

```
message skip('trim') >> sizeLessThan(1000)
```

Skip function must always lead subrules and not be preceded by converters or validators, otherwise a runtime error will be thrown.

8.17 Reusing rules

Any subrules composition can be stored in a variable and then used as a part of other rules. Such variables can be defined using the `@Rule` annotation, for example:

```
@Rule
toGroup = {trim >> {value -> Enum.valueOf(Groups, value)}}

// a rule that takes more than one parameter
@Rule
formatPost = {parameter, format -> trim >> stripTags >> format}

message formatPost({addSocialButtons(it)}) >> isLengthLess(100)
```

In a rule declaration the right side should be always a closure that takes a processed value parameter as the first argument and contains the rule itself as a closure body.

8.18 Rule extension

Any rule can be extended to perform additional processing of a parameter value. This might be helpful, for example, when there are several preprocessing stages or a certain constraints should be applied if, and only if, some condition is satisfied:

```
username isAlnum
if (!debug) {
  username isUniqueUsername
}
```

Here the `isUniqueUsername` validator is applied in addition to `isAlnum` only when the `debug` variable is coerced to `false` [80].

8.19 Multi-line rule

For better readability, a rule can span across multiple lines:

```
id trim >> ... >>
   splitTags >> maxLength(10)
```

To conform to Groovy syntax, while splitting make sure that the last token in each line is an operator.

8.20 Boolean converters

By default functions and closures that return values of type Boolean are evaluated as validation expressions, but you can easily tell the rule engine to treat them as converters by applying the tilde operator:

```
inverseParameter toBoolean >> ~{!it}
```

or

```
// inverse return type is Boolean
inverseParameter toBoolean >> ~inverse
```

Another way of implementing boolean converters is a `@Converter` annotation. It can be used for any method and exempts a developer from prefixing each validator call with the tilde-operator. For example:

```
@Converter
boolean inverse(boolean value) {
  !value
}

inverseParameter toBoolean >> inverse
```

8.21 Variables

Grules rules scripts support a variable declaration. The syntax is identical to what is used in usual Groovy scripts, for example:

```
a = 1
id gt(a)

male = ValidationUtils.MALE
gender eq(male)
```

8.22 Not logged rules

For some sensitive data, such as a credit card number or an answer to a security question, you may not want a parameter value to appear in a log file. In this case, you can use the `nolog` block and grules will never include values of the parameters from this block into a log record or send them to the client side. The following is one example of the use of the `nolog` block:

```
nolog {
  creditCard isCreditCard([CreditCard.VISA])
  password isValidPassword(username)
  ...
}
```

8.23 Validation block

Regardless of the DSL expressivity, you may still have preprocessing logic that cannot be defined as a grules rule. Such logic can be always placed in a `validate` block which should return a map of clean values processed by this block or throw a `ValidationException` if any of the values fail validation (you can find more about the exception in Chapter 10). For example:

```
GET:
id toInt
...
name trim

validate {
  def sum = ($options.collect { // options is an HTTP array parameter
    it == 'Yes' ? 1 : 0
  }).sum()
  if (sum <= 1) {
    throw ValidationException(NOT_ENOUGH_OPTIONS,
        'Please choose more options', 'options')
  }
  // return two clean parameters
  [options: $options, chosenOptionsNum: sum]
}

POST:
...
```

```
validate {
  ...
}
```

Chapter 9

Script Management

This chapter will provide an overview of the operations that can be applied to script files, including a description of how to reuse grules scripts and interact with rule engine API. We will also cover the conventions that should be followed to ensure the successful integration with a popular Grails Framework.

9.1 Script inclusion

Oftentimes you will find some common rules that might be shared between several scripts, so to not duplicate the code you can incorporate one rule script into another using the `include <RuleScriptClass>` statement. For example, if we have two forms: create and edit profile, they probably will be almost identical, so we can include common rules in both scripts.

We can place shared rules in *`ProfileGrules`*:

```
name isAlpha
email isEmail
...
```

Rules unique for a create profile page in *`CreateProfileGrules`*:

```
include ca.mcmaster.ProfileGrules

username isAlnum >> isUnique(username)
...
```

And rules unique for an edit profile page in *`EditProfileGrules`*:

```
include ca.mcmaster.ProfileGrules
```

```
userId trim >> toPositive
...
```

Also note that rules can be included in any part of the script, e.g.:

```
if (!debug) {
  include RegistrationSecurityGrules
}
```

For a functions script one can use a standard Java `import static` feature:

```
import static IntValidators.*

boolean isOdd(Integer num) {
  !isEven(num) // isEven is defined in IntValidators
}
```

9.2 Grails plugin

A grules plugin for Grails follows the convention over configuration design principle [123], which simplifies integration with the framework. By convention, rules scripts should be located in the same directory as a controller, and have the same name except the `Controller` part. So, for example, a rules script for `LoginController` should have a name `LoginGrules`. There are no any restrictions for path or name of function scripts.

9.3 Rule engine API

With grules you are able to apply preprocessing rules not only to request parameters but to any arbitrary map that contains variable names and their values. You can apply a rules script to such an input via the `Grules.applyGroupRules` method. The method takes a script class, a parameters map, and an optional environment functions and variables as its parameters. Here is an example of unit tests for a rules script that validate two parameters, `id` and `name` (for the sake of readability we use the Groovy test framework Spock [90]):

```
def "Test method for rules script with one group"() {
  when:
  def grules = new Grules() // for static calls use GrulesAPI class
```

```
  def r = grules.applyRules(MyFormGrules, [id:1, name:'Bob'])
  then:
  r.cleanParameters == [id:1, name:'Bob']
}

def "Check that rules are applied to group parameters"() {
  when:
  def r = grules.applyGroupRules(MyGrules, [GET: [id:1, name:'Bob']])
  then:
  r.cleanParameters == [GET: [id: 1, name: 'Bob']]
}
```

Similarly, you can call Grules from Java:

```
Map<String, Object> parameters = new HashMap<>();
parameters.put("id", new Integer(1));
grules.applyRules(MyGrules.class, parameters);
```

You can also pass additional custom functions as a third method argument:

```
def functions = [isJohn: {it == 'John'}]
def result = grules.applyRules(FunsGrules, [name:'Bob'], functions)
```

`FunsGrules` file*:*

```
name isJohn
```

Converters and validators can be tested as regular Groovy code, e.g.:

```
expect:
CommonFunctions.trim(' test') == 'test'
```

The other example where you might need to explicitly call the rule engine is for preprocessing of non-query string parameters. In addition to GET/POST request parameters and cookies, a part of the input data can be passed to an application via a URL path. For instance, the popular site *StackOverflow.com* uses a schema where a question id and canonical title are placed after a module name in a URL path. Here is a typical address of a question page:

```
http://stackoverflow.com/questions/111102/javascript-closures-work
```

Similarly to tampering with HTTP parameters, an attacker can perform a request manipulation and modify a URL to conduct a path traversal or injection attack. For example, in the URL above one can easily falsify the question id, and if the value is then inserted into the web page without any validation, site visitors may be exposed

to an XSS attack. To keep application data secure a site need to be protected from such evasion.

In grules preprocessing of values from a URL path can be performed by applying script rules to a URL_PATH group which includes path parameters that require validation. For our *Stackoverflow.com* example the code in a page controller might look like this:

```
def questionId = getQuestionId(url)
def questionTitle = getQuestionTitle(url)
def parameters = [URL_PATH: [id: questionId, title: questionTitle]]
def r = grules.applyGroupRules(QuestionGrules, parameters)
```

Then in the `QuestionGrules`:

```
id toPositiveInt
title match(/[\w\-]/)
```

As a future work, we are also going to parallelize processing of each parameter to leverage multi-core CPUs.

Chapter 10

Error Handling

In addition to guaranteeing data consistency, the integration of a validation model in a client-server application requires a system for reporting constraint violations to the user, indicating the origin of the violation in the UI with a sensible error message or styling of control elements [2]. In this chapter we will cover how these errors can be specified and then processed in an application that uses the grules rule engine.

10.1 Basic error handling

The framework contains an error handling mechanism that catches validation exceptions on the server side and coordinates their transfer to the client, where each error is subsequently processed by the client library. Validation error messages are defined on a subrule level which allows a more specific and thus more helpful description of failure:

```
parameterName converter1 *errorId1 >> validationRule1 [errorId2] >>
validationRule2 [errorId2] >> ...
```

You can use any Groovy object as an error id, for example:

```
nights trim >> isEven *'Number of nights cannot be odd'
```

To localize a message, you can use fields of the object `m` and the rule engine will read the property with the corresponding name from a resource bundle specified in a configuration file, for example:

```
nights isEven [m.notEven] // notEven property from a resource bundle
```

If no error message was defined for some subrule, then it is taken from the nearest subsequent subrule with a message, and if there is no such subrule, an empty

string is used instead. In this way the error message can cover multiple subrules simultaneously.

In order to define an error message that must be used when a required parameter is blank, you can specify a special property in the configuration file (`requiredErrorMessage`) with the error message or add a resource bundle properties of a form `required.<formName>.<parameterName>=<message>` in which case a custom error message will be used for each parameter.

10.2 Custom error parameters

In addition to an error message one can define custom error parameters that will be passed to a client side. For example, to change the input element next to which the error has to be shown an `element` parameter can be used:

```
pswdConfirm eq(pswd) [e('Passwords do not match', element: 'pswd')]
```

Here `e` is a method that takes an error message as the first argument, a map of custom parameters as the second, and constructs a validation exception that will be thrown in case of unsuccessful validation. If an error message is not needed the first parameter can be skipped:

```
pswdConfirm eq(pswd) [e(element: 'pswd')]
```

10.3 Validation exception

Besides returning the boolean value `false` from a validator, there is one more way to indicate invalid input — the `ValidationException`. This class contains several constructors: the first one takes a map with error properties (`Map<String, Object>`), the second, an error message represented by a `String` object, and the third constructor takes no parameters. You can pass any key/value pairs to the constructor, but the following error property names have a special meaning:

- `url` is a redirect URL
- `element` is the element next to which an error message will be shown
- `errorCode` is any object that identifies the error
- `message` is the error description

For example, the exception can be thrown from a converter to indicate that some precondition required for successful normalization of an input value is violated:

```
int toInt(String value) {
  try {
    Integer.valueOf(value)) {
  catch (NumberFormatException e) {
    throw new ValidationException(e.message)
  }
}
```

Another usage example would be a validator for password strength:

```
boolean isStrongPassword(String value, List<String> mediumRegexps,
    List<String> strongRegexps) {
  if (!mediumRegexps.every({value.matches(it)})) {
    throw new ValidationException(PasswordStrength:
        PasswordStrength.WEAK)
  } else if (!strongRegexps.every({value.matches(it)})) {
    throw new ValidationException(PasswordStrength:
        PasswordStrength.MEDIUM)
  } else {
    true
  }
}
```

All property values are converted to strings using the `toString()` object method before sending to a client.

10.4 Redirect URL

Apart from preprocessing values of form inputs, grules can be responsible for the validation of any request parameters. In that case, one may want to redirect a user to an error page when some request parameter fails validation. This behavior can be implemented by adding a `ONERROR_REDIRECT_URL` constant before rule definitions, for example:

```
ONERROR_REDIRECT_URL = 'http://example.com/error.html'
id isNaturalInt
email isEmail
...
```

or

```
ONERROR_REDIRECT_URL = Urls.ERROR_URL
id isNaturalInt
...
```

To override the error URL for a certain parameter you can use a custom property `url`, for example:

```
categoty isNaturalInt >> isCategory [e(url:Urls.WRONG_CATEGORY)]
```

10.5 Value placeholder

To include a current parameter value into an error message one can use an underscore token. For example:

```
date isDate [e(message: '_ is not a date')]
```

The token can be escaped by a backslash. Please note that to avoid injection attacks all HTML characters inside a parameter value are escaped before conveying an error message to a client (you can change this behavior in a configuration file).

10.6 Handling of grules exceptions in main application

Sometimes, passing the preprocessing barrier can be insufficient condition to consider a value valid, as part of the value's properties might be revealed only after business logic from a main application is applied. For example, a value can appear to be invalid in regard to the current model state or causes a runtime error. To communicate such an error to a client one can use the method `buildResponse(Map error)` from the `ValidationResponseHandler` class. This method builds a response that can be processed by the grules library on the client side. Here is a snippet that creates a validation response from a main application:

```
String getResponse() {
  ...
  if (!error) {
    // build confirmation page
  } else {
    return ValidationResponseHandler.buildResponse([
      // clean values
      [:],
```

```
      // invalid values
      POST: [
        // get the property invalidaCaptcha from the resource bundle
        captcha: ValidationResponseHandler.getErrorMessage(
            'invalidCaptcha')
        ]
      ]
    )
  }
}
```

10.7 Handling of non-validation exceptions

As grules is a Groovy internal DSL, it inherits its semantics including the exception handling machinery. Hence, a validator or converter can throw an exception that is not directly related to the input preprocessing rules but that still needs to be caught (for example, `IOException`). To handle such exceptions a developer can provide an implementation for a `handleNonValidationException` closure and point to it from the configuration file. The closure is called each time a DSL script triggers an exception that was not handled by the grules framework. Below is an example that shows a simple non-validation exception handler:

```
String handleException(Throwable e, String parameterName) {
  Logger.error(e.getMessage())
  return ValidationResponseHandler.buildErrorResponse([
    parameter: 'form',
    message: e.message])
  }
}
```

Chapter 11

Client Library

While most of the code needed to associate preprocessing rules with interface control elements is generated by the framework, some customization of client-side validation may still be necessary to enhance user experience. In this chapter we will describe the grules library that manages the rules on the client side and explain how to create handlers triggered on certain validation events.

11.1 Initialization

Each page that contains a form with preprocessing rules should include the grules client library; for example, it can be added in the HTML header section:

```
<head>
  ...
  <script src="/js/grules.js" type="text/javascript"></script>
<head>
```

This library contains the core functionality needed for managing form input, such as Ajax validation handlers, and implementation of the most common rules.

After an input form is processed by a browser rendering engine, the `Grules.init(config, formIds)` function should be called. The function fetches rules from a server, binds generated event handlers to form elements, and adds `required` [18], `maxlength` and other attributes to applicable inputs. For example, the initializer can be called on a `document.ready` event:

```
$(document).ready(function(){
  Grules.init(config, formIds);
  ...
});
```

The `config` parameter is a configuration object that has the following fields (each of them has a default value and can be left undefined):

- `locale` — a locale for an error messages resource bundle (defaults to `en`)
- `input<EventName>` — input validation handlers (see Section 11.3)
- `form<EventName>` — pre-submission validation handlers
- `inlineValidationEnabled` — a boolean value which controls whether inline validation is enabled (defaults to `true`)
- `keyUpDelay(milliseconds)` — the pause between a keystroke and value validation in case of an onchange event (defaults to `0`)

For example:

```
config = {
  locale: 'en',
  inputValidationError: onInputValidationError,
  formBeforeValidation: onFormBeforeValidation,
  inputErrorClear: onInputErrorClear,
  ...
};
```

You can change any of the parameters after initialization via a global `Grules` object — for example, the locale can be modified by calling the method `Grules.setLocale(locale)`.

The `formIds` parameter is optional and specifies that only forms with the given ids can be set up. If this parameter is omitted, all forms on the page will be processed by the client library. This parameter allows handling of dynamically created elements and pages with several forms where only some of them should be validated by grules.

11.2 In-line validation

To check if an entry violates any validation constraints, the client library delegates evaluation of a rules script to a preprocessing server: an asynchronous server request with form data is sent each time a user inputs a new field. Upon such a request, the server runs a rules script and responds with a detailed error report which is then parsed on the client side. This communication is implemented via Ajax technology that allows us to inform users about input errors without interrupting

their activity. It makes the validation process fairly transparent from a user's perspective as there is no difference between a validation error caught on the client and server side (except a slight delay in UI).

If you target mobile users, the amount of client-server traffic can be a crucial factor. To address this issue and improve feedback from a user interface, you can implement some preprocessing functions on the client side. In this way, it is possible to significantly reduce number of server requests.

Each time a validation event occurs the library looks for a method in the `Grules` object on the client side that satisfies the following requirements:

1. It has the same name as the function used on the server side.
2. It takes zero or one parameter (a processed parameter value).

In case of a composite rule, subrules are evaluated until a corresponding client function can not be found.

For example, consider the following grules function and the rule that uses it:

MathFunctions:

```
boolean isPrime(int n) {
  ! (2 ..< n).any { p -> n % p == 0}
}
```

MainGrules:

```
num toInt >> isPrime
```

Now we can implement the same logic in JavaScript:

```
Grules.isPrime = function(n) {
  if (isNaN(n) || !isFinite(n) || n % 1 || n < 2) {
    return "Not a number";
  }
  for (var i = 2; i < m; i++) {
    if (n % i == 0) return false;
  }
  return true;
}
```

For the given rule grules will use the JavaScript function `isPrime` instead of an asynchronous server request and in this way eliminate the handoff latency of client-server communication.

11.3 Client validation events

The framework manages preprocessing of input data both on the client and server sides. In addition, the grules JavaScript library already contains default event handlers that provide basic feedback to a user about failed and successful validations. Sometimes you may need to override these handlers to adjust the process for your design and functional needs, for example, you can define how error messages should be displayed on a web page or extend logic applied on a form submission event.

To perform actions before validation of an input value occurs or to handle a result returned by the rules engine, the following callbacks can be defined:

- `inputBeforeValidation(inputElement)` — the function is called before the DOM element `input` is checked by the rule engine. If it returns `false`, element validation is cancelled.

- `inputFailedValidation(inputElement, error)` — the function is called when a given input fails validation. `error` is an object with three fields: `message`, `errorName`, and `errorProperties`. `message` is an error description, `errorName` is a property name that represents the message in a resource bundle (`null` if the error message is defined as a string literal), and `errorProperties` is a string to string map with custom error parameters.

- `inputSuccessValidation(inputElement)` — the function is called if a value passes validation. This event can be used to hide content added by the `inputValidationError` function or display feedback about successful validation of the input field.

In the `inputBeforeValidation` function you can check if some property holds on each firing of a validation event. For example, if a client should go online to perform validation, you can use the following function:

```
function inputBeforeValidation(inputElement) {
  if (!window.navigator.onLine) {
    showNotification(messages[“browserOffline”]);
```

```
  }
  return false;
}
```

As an example of `inputFailedValidation/inputSuccessValidation` actions could serve logic that adds/removes a red border around a validated element. If the `inputFailedValidation` callback is not defined, a default error handler displays a message tooltip near the element (its position is absolute in order to not change a form's layout):

Figure 10: Error tooltip

To ensure that there are no race conditions all values are processed sequentially in order of the events that triggered validation.

11.4 Form submission

After a user submits a web form, the client library checks that no required fields are blank, and then, if there are no errors, it sends data to a server via XmlHttpRequest.

To control a form submission process one can define the following callbacks:

- `formBeforeValidation(formElement)` — the function is called before validation of a form represented by a `formElement` DOM element. If it returns `false`, form submission is cancelled.

- `formFailedValidation(formElement, errors)` — the function is called if a form fails validation. `errors` is a map where each key is a DOM element that failed validation (can be a form element) and a value is a corresponding `error` object (see the previous section). One of the use cases for this function would be displaying all error messages at a specific location on the page.

- `formSuccessValidation(formElement, xhr)` — the function is called if a form passes client and server validation without any errors. The second parameter is an XMLHttpRequest object [35] that contains data about a submission request and response. In the body of this function you can take any actions to indicate a successful result using a JSON object from a server response.

11.5 Validate function

By default, a field is validated on a blur event which is fired when a user tabs or clicks away from a control element, causing it to lose focus. To check an input value beyond this event you can use the `Grules.validateInput(id)` method which performs validation upon an element with the given id, returning `true` if its value contains a valid entry and an error object otherwise. Similarly, the `Grules.validateForm(id)` method validates a whole form and returns a map of error objects as a result.

11.6 Managing of validation events

To control on which events validation should be run or disable it for certain fields one can use the following methods in the global `Grules` object:

Name	**Description**
`disableValidation()`	Turns off client side validation for a current page.
`disableValidation(element)`	Turns off validation for the given input element or form.
`disableValidation(element, eventName)`	Turns off validation for the given input element on `eventName` event. Usage example: `disableValidation("ssn", "onchange")`
`enableValidation()`	Turns on validation for a current page (turned on by default).
`enableValidation(element)`	Turns on validation for the given input element or form.
`enableValidation(element, eventName)`	Adds validation for the given input element on event `eventName`. If any other callback is already bound to the event it will be called before validation.

You can use the special "onValueChange" event name in the enableValidation(element, eventName) method to validate an element each time its value is changed either by direct user input or from JavaScript code. The event can be helpful, for example, to validate a jQuery UI date picker since we lose focus on this element before any data is entered in it and thus an onBlur event cannot be used. Please note that some delay between a value change and validation may occur in case of binding to the onValueChange event.

11.7 W3C standards support

With increasing support of HTML5 by all major layout engines, leveraging built-in browser validation features that implement the new standard [23] in combination with h5Validate [78] fallback makes the client library much more practical. In addition, we use a jQuery framework to assure cross-browser compatibility of a DOM model [45], and its form plugin [46] to automate Ajax form submission.

11.8 Summary

As you can observe, the grules client library gives freedom to react differently for different validation events via a callback mechanism. It does not limit a developer in terms of actions that can be taken on each event and allows to implement UI feedback handlers using both a template engine and pure JavaScript DOM manipulation.

Chapter 12

Configuration

This chapter covers the format of the grules configuration file, all supported properties, as well as how to setup a logger used by the rule engine to record the results of rule application.

12.1 Configuration file format

We chose the ConfigSlurper [80] tree format for a grules configuration file as one that allows to use Groovy expressions as property values. File example:

```
log {
  dev {
    path = "create-drop"
    ...
  }
  ...
}
error {
  runtimeExceptionHandler = myproject.RuleEngineExceptionHandler
  ...
}
```

12.2 Properties

Here is a list of properties that can be defined in a configuration file:

- default preprocessing rules
- log settings

- actions to be taken for `@Security` rules
- a name of a resource bundle with error messages (`messages` by default)
- a maximal acceptable query string length
- a default parameters group
- a callback for tokens validation
- an uncaught exception handler
- an error message for blank but required fields
- a default date format

12.3 Logging

Grules can produce a log record with the results of rule evaluation for each parameter. By default it will log a client IP, validator name, value, query, and request time, but you can always adjust this output using configuration properties. In addition, you can specify a separate log file for requests that do not pass security checks. For example, you can have the following logging configuration:

```
Log.with {
  columns = [DATE_TIME, IP, VALUE, QUERY, DATE]
  securityFile = “/path/to/log/file”
  events = [SECURITY_FAIL, SUCCESS]
}
```

Chapter 13

Built-in Functions

The framework includes a library with validators and converters based on the most common parameters types and tasks related to normalization of their values. The list of included functions was built after careful analysis of many popular validation frameworks [1, 4, 29, 31, 57, 62 - 68, 73, 85, 121] and partly based on our own experience in development of online applications. The library is loosely coupled with other framework modules, which eases its integration into another project.

Reuse of the library code will ensure that rules are applied consistently, relieve developers from writing custom code for common preprocessing tasks, and reduce maintenance efforts.

13.1 Validators

Function name	**Requirement to processed value parameter**
`areIn(List values, List list)`	is a list which values are equal to any member of the specified list
`areIn(List values, Set set)`	is a list which values are equal to any member of the specified set
`contains(String value, CharSequence substring)`	contains the specified the specified sequence of char values
`endsWith(String value, String suffix)`	ends with the specified suffix

`hasNotCRLF(String value)`	does not contain a carriage return or a line feed character (see CRLF attack [26])
`isAfter(Date value, Date minDate)`	is a date after the specified one
`isAfterNow(Date value)`	is a date after now
`isAlnum(String value)`	is a string that matches `/[a-zA-Z0-9]+/`
`isAlnumSpace(String value)`	is a string that matches `/[a-zA-Z0-9\s]+/`
`isAlpha(String value)`	is a string that matches `/[a-zA-Z]+/`
`isAlphaSpace(String value)`	is a string that matches `/[a-zA-Z\s]+/`
`isAny(List values)`	is a list whose at least one member is true according to the Groovy Truth
`isAny(List values, Closure closure)`	is a list whose at least one member is valid according to a predicate
`isBefore(Date value, Date maxDate)`	is a date before the specified one
`isBeforeNow(Date value)`	is a date before now
`isBetween(Number value, Number min, Number max)`	falls within the specified range of decimal values
`isBetween(Integer value, IntRange min..max)`	falls within the specified integer range (uses HTML5 `min` and `max` attributes for the client side [18])
`isBirthDateAlive(Date value)`	is a date before the current year (or the current year) but not more than 120 years in the past [53]
`isBirthYearAlive`	is a year before the current year (or the current year) but not more than 120 years in the past
`isColor(String value)`	is a CSS color [22]

`isCountryCode(String value)`	is a country code [39]
`isCurrencyCode(String value)`	is a currency code (ISO 4217)
`isEmail(String value)`	is an email address compliant to RFC [21] (uses the HTML5 email input type for the client side)
`isEmpty(List values)`	is an empty list
`isEmpty(String value)`	is a string that matches `/\s*/`
`isEqual(value, object)` (alias `eq`)	is equal to the specified object
`isEven(Long value)`	is an even number
`isEvery(List values)`	is a list whose members are true according to the Groovy Truth
`isEvery(List values, Closure closure)`	is a list whose members are valid according to a predicate
`isFalse(value)`	coerces to boolean `false`
`isFileExtensionIn(FileItem value, List<String> extenstions)`	has an extension from the specified list (by default no extensions are allowed)
`isFileSizeLess(FileItem value, Integer size)`	is a file whose size is less than specified (in bytes)
`isGreater(Number value, Number number)` (alias – `gt`)	is a number greater than the specified number
`isGreaterEq(Number value, Number number)` (alias – `gte`)	is a number greater or equal to the specified number
`isHostname(String value)`	is a host name: DNS host name (e.g. bit.ly), IP address, or local host name (e.g. localhost)
`isIn(value, List list)`	is equal to any member of the specified list
`isIn(value, Set set)`	is equal to any member of the specified set

`isInternalUrl(String value)`	is a URL that belongs to the same domain as the current request (can be useful for redirect URL)
`isISBN(String value)`	is ISBN [64]
`isIPv4(String value)`	is an IP address version 4
`isIPv6(String value)`	is an IP address version 6
`isLengthEq(String value, Integer length)`	has the specified string length
`isLengthBetween(String value, Integer min, Integer max)`	is within the specified range of string lengths (the range is inclusive)
`isLengthLess(String value, Integer length)`	has a string length less than specified value
`isLengthLessEq(String value, Integer length)`	has a string length less or equal to specified value
`isLengthMore(String value, Integer length)`	has a string length more than specified value
`isLengthMoreEq(String value, Integer length)`	has a string length more than or equal to the specified value
`isLess(Number value, Number number)` (alias: `lt`)	is a number less than the specified number
`isLessEq(Number value, Number number)` (alias: `lte`)	is a number less than or equal to the specified number
`isMonth(Integer value)`	is an integer between 1 and 12
`isMonth(String value)`	represents a month for the current locale or is an integer between 1 and 12
`isName(String value)`	contains only characters that occur in human names, i.e. [77]: `[a-Z]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF]`

`isNotSqlInjection(String value)`	tests to see if the value appears to be an explicit SQL Injection attack (uses OWASP AppSensor [34])
`isOdd(Long value)`	is an odd number
`isPhone(String value)`	is an international phone number [77] (validator follows the forgiving format, thus parentheses, dots, spaces and hyphens are allowed)
`isPostcode(String value, String countryCode)` (alias `isZipcode`)	is a postcode valid for the specified country code (US and CA are supported [39])
`isSSN(String value, String countryCode)`	is social security number valid for specified country code [39]
`isState(String value)`	is a US or Canadian state or province in two letter format
`isStrongPassword(String value, List<String> mediumPasswordRegexes, List<String> strongPasswordRegexes)`	is compliant with a strong password policy A password is: - weak if it does not match some of `mediumPasswordRegexes` - medium if it does not match some of `strongPasswordRegexes` - strong if it matches all the given regexes Password is invalid if it is not strong, in this case `ValidationException`'s `parameters` field contains password strength (`WEAK` or `MEDIUM`), which then can be read on the client side
`isTimeZone(String value)`	is a ISO 8601 time zone [36]
`isTrue(value)`	coerces to boolean `true`
`isUrl(String value)`	is a well-formed URL [58]
`isWeekDay(String value)`	represents a day of the week for the current locale

matches(String value, String regexp)	is a string that matches the specified regex (uses HTML5 attribute pattern for the client side [25])
startsWith(String value, String prefix)	starts with the specified prefix

Credit card number:

- isCardNumber(String value, List<String> types)

  The validator checks a credit card number against the Luhn algorithm[49] and the specified list of type codes. The function follows the forgiving format pattern and allows to use spaces or hyphens in any place.

  Possible type codes: AMEX, BANKCARD, DINERS, DISC, ELECTRON, JCB, LASER, MAESTRO, MC, SOLO, SWITCH, VISA, VOYAGER.

13.2 Converters

Base type converters:

- toBigDecimal (uses current locale)
- toBoolean (returns a boolean value according to the Groovy Truth)
- toChar
- toEnum
- toDouble
- toFloat
- toInt
- toLong
- to<Type>List, where <Type> is any type that has a predefined converter

Other type converters:

- toNaturalInt (convert value to a non-negative integer number)
- toNaturalLong
- toNonnegativeBigDecimal

- `toPositiveInt`
- `toPositiveLong`
- `toPositiveBigDecimal`
- likewise, for float and double types

Date-time converters:

- ```
  toDate(String value, String pattern, Locale locale =
         Locale.default)
  ```

  The converter tests a date against the given date format using the default locale and time zone and creates a `java.util.Date` instance. Usage example:

  ```
  toDate('yyyyMMdd')
  toDate('MMMM, yyyy')
  ```

  The `pattern` parameter specifies the order in which the year, month and day values are passed (see the `java.text.SimpleDateFormat` class for the list of supported patterns). The converter can be used to process HTML5 input of type `datetime` [25, 70].

- `toMonthYear(String value)`

  The converter tests a month and year against the ISO 8601 using the default time zone, then transforms it to a date instance. The function can be used to process HTML5 input of type `month` [25, 70].

- `toTime(String value)`

  The converter tests a time against ISO 8601 (00:00:00 - 24:00:00), then transforms it to a date instance. The function can be used to process HTML5 input of type `time` [25, 70].

- `toWeekYear(String value)`

  The converter tests a week and year against ISO 8601, then transforms it to a date instance. The function can be used to validate HTML5 input of type `week` [25, 70].

Other converters:
  ```

Function name	**Action taken**
`abs(Integer/Long/Float/Double value)`	returns the absolute value of a int, long, float, or double value.
`add(Number value, Number number)`	adds `number` to the value (same as `{it + number}`)
`capitalize(String value)`	capitalizes the first letter
`ceil(Number value)`	rounds fractions up
`collect(List values, Closure transform)`	transforms each list member into a new value using the `transform` closure
`decimalFormat(String value, String pattern)`	calls `DecimalFormat.applyPattern` with the value and the pattern [97]
`div(Number value, Number number)`	divides the value by the given number
`escapeUrl(String value)`	escapes URL characters
`escapeHtml(String value)`	escapes HTML characters
`escapeJavascript(String value)`	escapes the characters using JavaScript string rules
`escapeSql(String value, org.owasp.esapi.codecs.Codec codec)`	escape input SQL characters, according to the selected codec (uses ESAPI [34]) This converter is not recommended, use prepared statements when possible [103].
`floor(Number value)`	rounds fractions down
`format(String value, String format, List args)`	formats the value using the specified format string and arguments, see `String.format(String format, Object... args)` for more details [97]
`mod(Long value, Long number)`	returns the value modulo `number`
`minus(Number value, Number number)`	subtracts `number` from the value (same as `{it + number}`)

`mult(Number value, Number number)`	multiplies the value by `number`
`replaceAll(String value, String regexp, String replacement)`	replaces each substring of the value that matches the given regular expression with the given replacement
`round(Number value)`	rounds to the nearest integer
`setLowerLimit(Number value, Number min)`	substitutes the value with `min` if it is less than `min`
`setUpperLimit(Number value, Number max)`	substitutes the value with `max` if it is greater than `max`
`sha512(String value, String salt)`	generates a SHA-512 [75] hash for the value prefixed by `salt`
`stripTags(String value)`	strips HTML tags (a default rule)
`substring(String value, int beginIndex)`	converts to a new string that is a substring of `value`; the substring begins at the specified `beginIndex` and extends to the end of the string
`substring(String value, int beginIndex, int endIndex = -1)`	converts to a new string that is a substring of the value; the substring begins at the specified `beginIndex` and extends to the character at index `endIndex - 1`
`subtract(Number value, Number number)`	subtracts `number` from the value
`toLowerCase(String value)`	converts the value to lowercase (using the default locale)
`toUpperCase(String value)`	converts the value to uppercase (using the default locale)

Chapter 14

Example

To assess the solution presented in this document, we have tried to implement validation for the NBC.com registration form in grules. This example will cover conversion and validation use cases at a basic level and give you a flavour of what can be done using our framework.

The registration process at NBC consists of two stages. Initially, a user has to enter his email address:

Figure 11: NBC.com Sign Up Form, step 1

After the user enters the email address, a few more fields, such as a username and password, must be filled in:

Figure 12: NBC Sign Up Form, step 2

Now, for the sake of simplicity, let's assume that HTML for the first form looks like this:

```
<form data-token="6bae9a6c8a8c06cbda25c12c6af48517fbc2e00b...">
  <input type="email" name="email" class="email"/>
  <input type="submit" value="Sign Up" class="submit"/>
</form>
```

The validation script for this step would then be pretty trivial and have just three lines of code.

SignUpStageOneGrules:

```
package rules

import static rules.SignupFunctions.*

email isEmail [m.invalidEmail] >> isUniqueEmail [m.notUniqueEmail]
```

Here, `invalidEmail` and `notUniqueEmail` are properties from the messages file, and `isUniqueEmail` is a custom function defined in the *`SignupFunctions`* file:

```
package rules

import dao.UserDao // A data access object class for a user entity

@Functions
class SignupFunctions {
```

```
  boolean isUniqueEmail(email) {
    !UserDao.where(email: email)
  }

  ...
}
```

An alternative solution would be to call the `UserDao.where(email: email)` method as a closure:

```
email isEmail [m.invalidEmail] >>
      !{UserDao.where(email: it)} [m.notUniqueEmail]
```

At the second stage of form validation, we need to check each input, including the hidden one with the email address.

SignUpStageTwoGrules:

```
package rules

import static rules.SignupFunctions.*
// include the rules script from the first stage to check the email
include rules.SignUpStageOne

username isAlnum [m.invalidUsername]

password isStrongPassword([/.*\w.*/, /.*\d.*/, /.{5,}/],
         [/.*\+.*/]) [m.weakPassword] >>
         isLengthLess(20) [m.passwordTooLong]

month isMonth [m.invalidMonth]

day toInt >> isDay [m.invalidDay]

year toInt >> toFourDigitYear >> isBirthYearAlive [m.invalidYear]

@Parameter
date = createDate(year: year, month: month, day: day) [m.InvalidDate]

gender toEnum(Gender) [m.wrongGender]

postCode['00000'] isPostCode('US') [m.invalidPostCode]

termsConditionsCheckbox toBoolean
```

```
if (termsConditionsCheckbox) {
  subscribeCheckbox toBoolean
}
```

Then in *SignupFunctions*:

```
/**
 * Converts a two-digit year to a four-digit year,
 * assuming that we operate with a date that falls
 * between last 99 years and the current year.
 */
int toFourDigitYear(int twoDigitsYear) {
  if (!(twoDigitsYear in 0..99)) {
    throw new ValidationException('Invalid year')
  }
  def year = Calendar.instance.get(Calendar.YEAR)
  def firstCenturyYear = year - year % 100
  if (twoDigitsYear > year % 100) {
    firstCenturyYear - 100 + twoDigitsYear
  } else {
    firstCenturyYear + twoDigitsYear
  }
}
```

An attentive reader will see that the rules script code is self-explanatory but dense, especially in comparison with the amount of code that needs to be written in Java to implement the same logic. These language features ensure that complexity of preprocessing code is reduced by a significant factor, which makes the application easier to maintain and increase its ability to evolve.

Chapter 15

Conclusion and Future Work

15.1 Conclusion

In this thesis, we have argued that absence of a deliberate way to preprocess external data in large client-server applications causes complexity, security, and user experience problems. In order to attempt to resolve these issues, we proposed a technique that considers input preprocessing as an independent stage and that allows to express rules which must be applied to external data via the domain-specific language. We have shown that it solves the aforementioned problems more effectively than unstructured adoption of any single general purpose language or a declarative language like XML, and in making this argument we briefly surveyed existing common approaches to input validation, paying attention to the strengths and weaknesses of each method. By providing a working prototype, we have proven that following the reasons mentioned in this document, building a rules engine that is based on a DSL that supports syntax and semantics of the host application's language, has built-in preprocessing functions and a front-end library would be a tangible step in a direction of less complex, but more secure, scalable, and user friendly applications.

Despite the fact that we used a web site as an example, the DSL is equally applicable to other domains like mobile software or enterprise bank systems. grules provides the rules semantics that can be leveraged in various ways to support a broad range of preprocessing scenarios: from simple string trimming to complex conditions that protect an application from hacker attacks. The proposed framework allows for a quick and easy start during the prototyping stage (rich converters and validators library, out of the box event handlers), as well as complex data preprocessing in a production environment (custom functions, internalization, customization of error feedback). With DSL scripts, dispersion of input requirements throughout the codebase is minimized as they can now be separated from business logic and organized according to the subdomains to which they apply. Finally, by using JVM as an execution platform, grules benefits from the existing community and code

libraries, while implementing it as a Groovy DSL allows seamless integration of the rule engine into an existing project.

However, all these strengths do not come without limitations. The main drawbacks are a lack of complete control over expressions used in the rules scripts and the exceptions reporting apparatus that operates at a lower level and does not always take the DSL semantics into account.

Nevertheless, we observe an appreciable trend in the software engineering community toward such solutions [19, 20] and hope that our work would be useful to others studying the same problems in data processing.

15.2 Future work

In our project we used asynchronous XMLHttpRequest to implement inline validation on the client side. This approach allows to avoid code duplication but imposes some performance overhead which can lead to delays in the user interface (especially on mobile platforms). In order to mitigate this issue, a key goal for future work is to implement a Groovy to JavaScript translator that can convert simple expressions, such as constants and arithmetic operations, to corresponding client code. In this way we would be able to significantly reduce the number of client/server requests needed for the inline validation.

Appendix

Core rules script grammar

Notations:

- `.r` — regular expression
- `{}` — zero or more occurrences
- `[]` — optional
- ‘’ — string literal

```
name ::= ‘[a-zA-Z_]\w*’.r

error ::= ‘[‘ GroovyExpression ‘]’

subrule ::= (conversion_subrule | validation_subrule) [error]

rule_exp ::= subrule | rule_exp ‘>>’ subrule

rule_application ::= param_name [‘[‘ GroovyExpression ‘]’] rule_exp

conversion_subrule ::= [‘~'] (function_call | GroovyClosure)

validation_subrule ::= validation_factor ‘||’ validation_factor |
validation_factor

validator_factor ::= validator_factor ‘&&’ validator_factor |
‘!’ validator_factor | function_call | GroovyClosure | ‘(‘
validation_exp ‘)’

exp_list ::= GroovyExpression {‘,’ GroovyExpression}

function_call ::=  function_name [`(´ [exp_list] `)´]
```

Bibliography

[1] Ed Burns editor, *JavaServer™ Faces Specification Version 2.2*, Oracle America Inc, June 2011

[2] Danny M. Groenewegen, Eelco Visser, *Integration of data validation and user interface concerns in a DSL for web applications*, Delft University of Technology, Software Engineering Research Group, 2010

[3] D. Crockford, *The application/json Media Type for JavaScript Object Notation (JSON)*, Network Working Group, July 2006

[4] David Sawyer McFarland, *JavaScript & JQuery: The Missing Manual,* 2nd Edition, O'Reilly Media, Inc., November 2011

[5] Steven C. McConnell, *Code complete (2nd Edition)*, Microsoft Press, July 2004

[6] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, T*he Java™ Language Specification,* 3rd Edition, Addison-Wesley, May 2005

[7] Ralph Gordon, Walter Egan, Peter Jew, Kathryn Munn, Landon Ott, Robin Whitmore, *Oracle® Fusion Middleware*, Oracle, April 2010

[8] Mike Grogan, *JSR-223 Scripting for the Java TM Platform*, Sun Microsystems, December 2006

[9] Fergal Dearle, *Groovy for Domain-Specific Languages*, Packt Publishing, June 2010

[10] Jenifer Tidwell, *Designing Interfaces,* 2nd Edition, O’Reilly, December 2010

[11] Paco Hope, Ben Walther, *Web Security Testing Cookbook*, O'Reilly Media Inc., October 2008

[12] Stuart McClure, *Hacking Exposed: Network Security Secrets & Solutions*, Sixth Edition, McGraw-Hill Osborne Media, January 2009

[13] Kristina Boysen Taylor, *A specification language design for the Java Modeling Language (JML) using Java 5 annotations*, Iowa State University, 2008

[14] Elizabeth Fong and Vadim Okun, *Web Application Scanners: Definitions and Functions*, Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, 2007

[15] S. Ruby, D. Thomas, and D. Heinemeier Hansson, *Agile Web Development with Rails*, Third Edition. Pragmatic Programmers, 2009

[16] Ben Moseley, Peter Marks, *Out of the Tar Pit*, February 2006.

[17] *Inline Validation in Web Forms*, *http://www.alistapart.com/articles/inline-validation-in-web-forms/* (last visited May 2011)

[18] Michael Smith, *HTML: The Markup Language*, Unofficial Editor's Draft 15 February 2012, *http://dev.w3.org/html5/markup/*

[19] *How to use the validation rules on both client side and server side, http://stackoverflow.com/questions/2606283* (last visited September 2011)

[20] *JavaScript validation issue with international characters, http://stackoverflow.com/questions/1073412* (last visited September 2011)

[21] *RFC-compliant email address validator*, *http://blog.dominicsayers.com/2009/01/28/rfc-compliant-email-address-validator/* (last visited September 2011)

[22] *Color state*, *http://www.whatwg.org/specs/web-apps/current-work/multipage/number-state.html* (last visited September 2011)

[23] Mark Pilgrim, *Dive into HTML5*, *http://diveintohtml5.appspot.com* (last visited September 2011)

[24] J. Klensin, *Simple Mail Transfer Protocol*, Network Working Group *http://tools.ietf.org/html/rfc5321*, October 2008

[25] Ian Hickson, HTML5, *A vocabulary and associated APIs for HTML and XHTML*, W3C Working Draft 25 May 2011, *http://www.w3.org/TR/html5/*

[26] *CRLF Injection attacks and HTTP Response Splitting*, *http://www.acunetix.com/websitesecurity/crlf-injection.htm* (last visited September 2011)

[27] *ASP.NET and Struts: Web Application Architectures*, *http://msdn.microsoft.com/en-us/library/aa478961.aspx* (last visited September 2011)

[28] Michael Scheidell, *Intrusion Detection System*, Patent US20100100961, April 2010

[29] *Struts Validator Guide*, *http://struts.apache.org/1.x/faqs/validator.html* (last visited September 2011)

[30] *Business rules engine*, *http://en.wikipedia.org/wiki/Business_rules_engine* (last visited May 2011)

[31] *Design Guidelines for Secure Web Applications*, Microsoft, MSDN, *http://msdn.microsoft.com/en-us/library/ff648647.aspx* (last visited June 2012)

[32] Chen Lei, *DoS and DDoS Attack's Possibility Verification on Streaming Media Application*, Software Sch., FUDAN Univ., Shanghai, 2008

[33] Robert A. Martin, *Common Weakness Enumeration Version 1.3*, The MITRE Corporation, 2009

[34] *The Open Web Application Security Project*, *https://www.owasp.org* (last visited October 2011)

[35] Anne van Kesteren, *XMLHttpRequest Level 2*, W3C Working Draft 17 January 2012,
http://www.w3.org/TR/XMLHttpRequest/

[36] *Time zone abbreviations*, *http://www.timeanddate.com/library/abbreviations/timezones/* (last visited October 2011)

[37] Walter L. Hürsch and Cristina Videira Lopes, *Separation of Concerns*, 1995

[38] Alexsander Reelsen, *Play Framework Cookbook*, Packt Publishing, Birmingam, Mumbai, July 2011

[39] *Country names and code elements*, ISO, 2011, *http://www.iso.org/iso/country_codes/iso_3166_code_lists/country_names_and_code_elements.htm*

[40] Louridas, P., *Static code analysis*, Greek Res. & Technol. Network, August 2006

[41] Maggie Johnson, *Syntax Directed Translation*, Summer 2008.

[42] Nikolay Georgiev, *A Web-Based Environment for Learning Normalization of Relational Database Schemata*, September 2008

[43] *ASP.NET Validation in Depth*, *http://msdn.microsoft.com/en-us/library/aa479045.aspx* (last visited October 2011)

[44] Alexander Kolesnikov, *Tapestry 5: Building Web Applications,* Packt Publishing, UK, January 2008

[45] *jQuery API*, *http://api.jquery.com*, 2010 The jQuery Project

[46] *jQuery Form Plugin*, *http://jquery.malsup.com/form/* (last visited October 2011)

[47] Alessandro Basso, *Protecting Web resources from massive automated access*, University of Torino, Italy, 2008

[48] *Scalaz: Type Classes and Pure Functional Data Structures for Scala http://code.google.com/p/scalaz/* (last visited October 2011)

[49] Chiyuan Li, Zhiqiang Yao, *The Validation of Credit Card Number on Wired and Wireless Internet*, North China Institute of Aerospace Engineering, Langfang, China, March 2011

[50] *Comparison of web browsers*, *http://en.wikipedia.org/wiki/Comparison_of_web_browsers* (last visited November 2011)

[51] *Web Form Validation: Best Practices and Tutorials*, *http://www.smashingmagazine.com/2009/07/07/web-form-validation-best-practices-and-tutorials/* (last visited November 2011)

[52] K. Murchison, RFC-5233, *Sieve Email Filtering: Subaddress Extension*, Network Working Group, Carnegie Mellon University, January 2008, *http://tools.ietf.org/html/rfc5233*

[53] Robine, J.-M., Allard, M., *The oldest human,* France, Science 279, 1998

[54] Tomaz Kosar, Pablo Martınez Lopez, Pablo Barrientos, *A preliminary study on various implementation approaches of domain-specific language*, University of Maribor, Slovenia, Universidad Nacional de La Plata, Facultad de Informatica, Argentina, April 2007

[55] M. Bravenboer, R. Vermaas, J. Vinju, E. Visser, *Generalized type-based disambiguation of meta programs with concrete object syntax*, Fourth International Conference on Generative Programming and Component Engineering, Springer-Verlag, 2005

[56] Martin Fowler, *Language Workbenches: The Killer-App for Domain Specific Languages*, 12 Jun 2005

[57] Emmanuel Bernard, *JSR 303: Bean Validation*, Red Hat Middleware LLC, 2007 November 14, *http://jcp.org/en/jsr/detail?id=303*

[58] JBoss community, *Hibernate Validator*, *http://www.hibernate.org/subprojects/validator.html* (last visited November 2011)

[59] Juergen Hoeller, Alef Arendsen, Colin Sampaleanu, Rob Harrop, Thomas Risberg, *Overview of Spring Framework 3.1*, Rod Johnson, 2011 SpringSource

[60] *WebDSL*, *http://webdsl.org/home* (last visited November 2011)

[61] *The Java™ Virtual Machine Specification*, 1999 Sun Microsystems, *http://java.sun.com/docs/books/jvms/second_edition/html/Instructions2.doc6.html* (last visited November 2011)

[62] *HTML5Pattern*, *http://html5pattern.com/* (last visited November 2011)

[63] *The Seam Framework — Next generation enterprise Java development*, *http://seamframework.org/* (last visited November 2011)

[64] *Zend Validate*, Zend Technologies Ltd., 2006 - 2011, *http://framework.zend.com/manual/en/zend.validate.html*

[65] *Pipeline (Unix)*, *http://en.wikipedia.org/wiki/Unix_pipe* (last visited June 2012)

[66] The PHP Group, *Data Filtering*, *http://php.net/filter* (last visited November 2011)

[67] Antti Valmari, *The State Explosion Problem*, Lecture Notes in Computer Science, Tampere University of Technology, Finland, 1998

[68] Jeffery Winesett, *Agile Web Application Development with Yii 1.1 and PHP5*, Birmingham, Mumbai, 2010 Packt Publishing

[69] Guillaume Laforge, *What’s new in Groovy 2.0*, 33rd Degree Conference, Krakow, Poland, March 2012

[70] *Regular Expressions Cookbook*, O'Reilly Media, Inc., May 2009

[71] *Hibernate Validator*, JSR 303 Reference Implementation Reference Guide, 4.0.1.GA, 2009 Red Hat Middleware, LLC. & Gunnar Morling

[72] *Simple Made Easy*, Rich Hickey, Oct 20, 2011, © C4Media

[73] Adrian Holovaty, Jacob Kaplan-Moss, *The Definitive Guide to Django: Web Development Done Right,* 2nd Edition, Springer, 2009

[74] Adam Griffiths, *CodeIgniter 1.7 Professional Development*, Packt Publishing, 2010

[75] Tom St Denis, Syngress, *Cryptography for Developers*, January 2007

[76] *Programming the Mobile Web: Ajax Support*, *http://programming4.us/mobile/2010.aspx* (last visited November 2011)

[77] *jQuery plugin: Validation*, *http://bassistance.de/jquery-plugins/jquery-plugin-validation/* (last visited November 2011)

[78] Eric Hamilton, *h5Validate — HTML5 Form Validation for jQuery*, *http://ericleads.com/h5validate/ (*last visited November 2011)

[79] *DSL Descriptors for Groovy-Eclipse, http://groovy.codehaus.org/DSL+Descriptors+for+Groovy-Eclipse* (last visited November 2011)

[80] *Groovy Language Specification*, SpringSource, *http://groovy.codehaus.org/jsr/spec/* (last visited December 2011)

[81] Dave Thomas, David Heinemeier, *Agile Web Development with Rails*, 2nd Edition, Hansson Pragmatic Programmers 2006

[82] *jQuery 1.3 with PHP*, Kae Verens, Birmingham, Mumbai, 2009 Packt Publishing

[83] Jodi Forlizzi, Katja Battarbee, *Understanding Experience in Interactive Systems*, Carnegie Mellon University, in Proceedings of DIS 2004 (Designing Interactive Systems)

[84] Konstantin Kafer, *Cross Site Request Forgery*, Hasso-Plattner-Institut, Potsdam

[85] François Zaninotto, Fabien Potencier, *A Gentle Introduction to symfony 1.4*, Sensio SA SA, May 2010

[86] *Symfony Jquery form validation plugin*, *http://www.symfony-project.org/plugins/sfJqueryFormValidationPlugin*, Symfony (last visited December 2011)

[87] Wadler, Philip, *Comprehending Monads*, 1990 ACM Conference on LISP and Functional
Programming, Nice, 1990

[88] Brad Arkin, Scott Stender, Gary Mcgraw, *Software Penetration Testing*, Security & Privacy, IEEE, 2005

[89] *Regular Expressions*, *The Single UNIX Specification*, The Open Group, 1997

[90] Peter Niederwieser, *Spock – the enterprise ready specification framework*, *http://code.google.com/p/spock/*

[91] Rodrigo Werlinger, Kirstie Hawkey and Konstantin Beznosov, *An integrated view of human, organizational, and technological challenges of IT security management*, University of British Columbia, Vancouver, Canada, November 2008

[92] Clifford Lynch, *Canonicalization: A Fundamental Tool to Facilitate Preservation and Management of Digital Information*, Coalition for Networked Information, D-Lib Magazine, 1999

[93] Almudena Alcaide Raya, Jorge Blasco Alis, Eduardo Galán Herrero, Agustín Orfila Diaz-Pabón, *Cross-Site Scripting*, University Carlos III of Madrid, Spain, 2011, IGI Global

[94] Jose Maria Alonso, Antonio Guzman, Marta Beltran, Rodolfo Bordon, Rey Juan, *LDAP Injection Techniques*, Informatica 64, Carlos University, Madrid, Spain, July 5, 2009

[95] William M. Farmer, *The Seven Virtues of Simple Type Theory*, McMaster University, December 2007

[96] *Web/Libraries/Formlets*, HaskellWiki, *http://www.haskell.org/haskellwiki/Formlets*, (last visited December 2011)

[97] *Java Platform Standard Edition 7 Documentation*, 2011, Oracle and/or its affiliates

[98] J.D. Meier, Alex Mackman, Blaine Wastell, Prashant Bansode, Andy Wigley, *How To: Use Regular Expressions to Constrain Input in ASP.NET*, Microsoft Corporation, May 2005

[99] J.D. Meier, Alex Mackman, Blaine Wastell, Prashant Bansode, Andy Wigley, *How To: Prevent Cross-Site Scripting in ASP.NET*, Microsoft Corporation, May 2005

[100] R. Snake, *XSS (Cross Site Scripting) Cheat Sheet*, *http://ha.ckers.org/xss.html* (last visited December 2011)

[101] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso, *A Classification of SQL Injection Attacks and Countermeasures*, College of Computing, Georgia Institute of Technology

[102] Jeongseok Seo, Han-Sung Kim, Sanghyun Cho and Sungdeok Cha, *Web Server Attack Categorization based on Root Causes and Their Locations*, Division of Computer Science, Department of EECS, KAIST and AITrc/IIRTRC/SPIC, 2004

[103] Stephen Thomas and Laurie Williams, *Using Automated Fix Generation to Secure SQL Statements*, Department of Computer Science, North Carolina State University, USA, 2007

[104] J.D. Meier, Alex Mackman, Michael Dunner, Srinath Vasireddy, Ray Escamilla and Anandha Murukan, *Improving Web Application Security: Threats and Countermeasures*,
Microsoft Corporation, June 2003

[105] Alan Shalloway, James Trott, *Design Patterns Explained: A New Perspective on Object-Oriented Design*, 2nd Edition, Addison-Wesley Professional, 2004

[106] David Hook, *Beginning Cryptography with Java*, Wrox Press, 2005

[107] *Java™ Platform, Enterprise Edition 6*, API Specification, Oracle Corporation and/or its affiliates, Generated on February 2011

[108] Grules. Rule engine for input preprocessing. *http://grules.org*

[109] *GEP 8 — Static type checking, http://docs.codehaus.org/display/GroovyJSR/GEP+8+-+Static+type+checking* (last visited October 2011)

[110] *Measuring Website Security: Windows of Exposure,* 11th Edition, WhiteHat Security Inc., December 2011

[111] Andrew Troelsen, *Pro C# 2010 and the .NET 4 Platform*, Apress, May 2010

[112] *Jeff Moser, Does Your Code Pass The Turkey Test? http://www.moserware.com/2008/02/does-your-code-pass-turkey-test.html* (last visited January 2012)

[113] Social Security Online, *Invalid or impossible Social Security numbers* (updated 09/21/2011), *http://ssa-custhelp.ssa.gov/app/answers/detail/a_id/425*

[114] *The Validation Application Block*, Microsoft, MSDN *http://msdn.microsoft.com/en-us/library/ff664356* (last visited January 2012)

[115] Brett Stineman, *IBM WebSphere ILOG Business Rule Management Systems: The Case for Architects and Developer*, IBM Software Group, November 2009

[116] Michal Bali, *Drools JBoss Rules 5.0 Developer's Guide*, Packt Publishing, July 2009

[117] MVEL, *Language Guide for 2.0*, The Codehaus, *http://mvel.codehaus.org/Language+Guide+for+2.0* (last visited January 2012)

[118] *Business rule management system*, *http://en.wikipedia.org/wiki/BRMS* (last visited January 2011)

[119] Vidyasagar Potdar, Farida Ridzuan, Pedram Hayati, Alex Talevski, Elham Yeganeh, Nazanin Firuzeh, and Saeed Sarencheh, Spam 2.0: The Problem Ahead, *Anti Spam Research Lab*, Curtin University of Technology, Australia, Springer-Verlag Berlin Heidelberg 2010

[120] J.A. Bargas-Avila, O. Brenzikofer, S.P. Roth, A.N. Tuch, S. Orsini and K. Opwis, *Simple but Crucial User Interfaces in the World Wide Web: Introducing 20 Guidelines for Usable Web Form Design*, University of Basel, Faculty of Psychology, Switzerland, 2011

[121] Bashar Abdul-Jawad, *Groovy and Grails Recipes*, Apress, December 2008

[122] *Google Trends*, Google Inc., *http://www.google.com/trends/* (last visited January 2012)

[123] *Convention over configuration*, *http://en.wikipedia.org/wiki/Convention_over_configuration* (last visited May 2012)

[124] The Dart Team, *Dart Programming Language Specification* (Draft Version 0.07, January 20, 2012), *http://www.dartlang.org/docs/spec/dartLangSpec.pdf*

[125] Daniel J. Barrett, *MediaWiki*, O'Reilly Media, Inc., 2008

[126] U.F. Yergeau, *UTF-8, a transformation format of ISO 10646*, Network Working Group, November 2003