An algorithm and architecture for server-side processing of model-bound compositional web forms

BACHELOR THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software Engineering

by

Bernhard Schussek

Registration Number 0625593

elaborate at the
Institut of Computer Aided Automation
Research Group Industrial Software
to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Thomas Grechenig
Assistance: Johannes Harms

Vienna, January 18, 2013
Abstract

Web forms that support rich interaction patterns are time-consuming to develop. This is mainly because of their inherent complexity and the lack of reusability and compositionality in many of today’s form libraries. This thesis proposes an algorithm for server-side form processing which addresses four challenges: Reusability of form components, the ability to compose components for modeling higher-level components, the reduction of crosscutting effects between the different layers of a Model-View-Controller architecture and the integration of the domain model into the form layer. Methodologically, this work uses an exemplary form to highlight the aforementioned challenges and then presents a three-step approach for solving them. The first of these steps is described in detail. It introduces an object-oriented framework consisting of form objects, data transformers, data mappers and event listeners. With these tools, one can construct reusable form components and exploit synergy effects between applications that employ similar interaction patterns. Together with a full implementation in the web framework Symfony2, the proposed algorithm is a significant step towards rapid development of rich web forms.

Keywords

Web Programming, Forms, Reusability, Compositionality, Domain Model
Kurzfassung


Schlüsselwörter

Webprogrammierung, Formulare, Wiederverwendbarkeit, Komposition, Domänenmodell
## Contents

### 1 Introduction
- 1.1 Motivation .................................................. 1
- 1.2 Contribution ................................................. 2
- 1.3 Outline ....................................................... 3

### 2 Background
- 2.1 Web Forms ..................................................... 4
- 2.2 PHP ............................................................ 5
- 2.3 Crosscutting .................................................... 6
- 2.4 Reusability ..................................................... 7
- 2.5 Model Binding .................................................. 7
- 2.6 Related Work .................................................. 8
- 2.7 Terminology .................................................... 10

### 3 Problem Specification
- 3.1 Introductory Example ......................................... 11
- 3.2 Generic Problems .............................................. 13
- 3.3 HTML Abstraction Problems ................................. 15
- 3.4 Field Abstraction Problems ................................ 16

### 4 Concept and Implementation
- 4.1 Three-Step Approach ......................................... 18
- 4.2 Form Model ..................................................... 19
- 4.3 Data Transformation .......................................... 21
- 4.4 Form Naming and Binding ................................... 25
- 4.5 Data Mapping .................................................. 27
- 4.6 Events .......................................................... 30
- 4.7 Summary ....................................................... 32

### 5 Conclusion
- 5.1 Results ......................................................... 35
- 5.2 Discussion ..................................................... 35
- 5.3 Usage Example ................................................ 36
- 5.4 Future Work ................................................... 39
- 5.5 Conclusion ..................................................... 40

### List of Figures

### List of Tables

### List of Listings

### Bibliography
Chapter 1. Introduction

1 Introduction

In recent decades, the World Wide Web (WWW) [7] has become an increasingly important source of information. For example, when internet users are looking for information, they consult a search engine such as Google\(^1\) or Yahoo\(^2\). When they want to learn some rough facts about a topic, they read them up on Wikipedia\(^3\), an online encyclopedia offering more than four million English articles at the time of this writing. When they want to know what your friends did the last weekend, they check out their profiles on Facebook\(^4\). When they want to know who won the elections, they don’t need to turn on the TV or the radio. Major broadcasting companies like the BBC\(^5\) or CNN\(^6\) have long moved their information channels to the web.

But the web today is not only about information, it is about participation. Many platforms provide simple ways to publish your own contents. YouTube\(^7\) made it trivial to share videos. When you want to publish a picture, go to flickr\(^8\) and you can do so within minutes. And on Blogger\(^9\), you can create your own personal online diary in no time.

The one major technology that enabled this trend was the Common Gateway Interface (CGI) [33], a protocol standard for exchanging data with a web server based on the Hypertext Transfer Protocol (HTTP) [18]. CGI, in combination with web forms, made it possible to submit information to the web from within your browser, while visiting a website. No longer did you have to learn about how to create websites and about the Hypertext Markup Language (HTML) [6], no longer did you have to rent webspace when you wanted to put information on the web. Just fill out a form, press a button and you are part of the game.

1.1 Motivation

While web forms made it very easy to interact with web applications, they also made them more complicated to develop. Advanced interaction patterns using technologies such as AJAX [21] require considerable time to implement. Repeated password entry, spam prevention via CAPTCHA inputs [1] or the selection of coordinates from an interactive map are common User Interface (UI) patterns seen on many websites, but reusing these patterns – once developed – is difficult and often impossible.

The lack of reusability necessarily leads to higher costs, lower productivity, a decreased software quality and potential security issues in the final product [34]. Furthermore, the nature of forms leads to crosscutting [27], that is, the duplication of structural information in different layers of an application. This structural duplication makes it impossible to change one layer of an application without having to adapt other layers as well. Essentially, the structure of a form

\(^1\) http://www.google.com \\
\(^2\) http://www.yahoo.com \\
\(^3\) http://www.wikipedia.org \\
\(^4\) http://www.facebook.com \\
\(^5\) http://www.bbc.co.uk \\
\(^6\) http://edition.cnn.com \\
\(^7\) http://www.youtube.com \\
\(^8\) http://www.flickr.com \\
\(^9\) http://www.blogger.com
duplicates the structure of the application’s domain model. So reducing crosscutting necessarily involves the integration of the domain model with the modeling of a form’s structure.

Obviously, these ideas are not new. On the client side, various industrial solutions exist for building forms out of reusable components. Two such libraries are Sencha JS\textsuperscript{10} and Dojo\textsuperscript{11}. While these libraries provide a convenient user and developer experience, they also rely on JavaScript being supported by the web browser, which is not always the case. Also, they cannot replace proper server-side handling and validation of the form input without jeopardizing the application’s security. Consequently, even though client-side libraries exist, they require fitting server-side counterparts.

On the server side, the reusability and crosscutting concerns have also been approached by various projects in the industry and the academia [2, 8, 14, 24, 32]. While these projects solve many important problems, few of them present an architecture that allows to create reusable components that are fit for the requirements of modern web applications. Even less allow to integrate the domain model into the form layer in order to reduce crosscutting to a minimum.

### 1.2 Contribution

The specific aim of this work is to identify and describe an architecture that:

- maximizes reusability of forms by abstracting their individual components,
- allows to compose such components to create new components of a higher complexity,
- reduces the effects of crosscutting to a minimum and
- provides an integration with the application’s domain model (model binding).

As a foundation for this architecture, the thesis provides a catalog of specific architectural problems that need to be solved. These problems are obtained by analyzing the low-level generation and processing code of an arbitrary form.

The thesis then presents a three-step approach to solving these problems and reaching the above goals. Each of these steps solves a particular set of problems and builds upon the previous steps. The final result is an architecture that can be implemented using different programming languages such as Java, Ruby, or PHP.

The first step of the approach is explained in detail. In this step, a generic form model is designed that is capable of representing arbitrary forms. The concepts of data mappers, data transformers and events are explained. These concepts, together with a generic algorithm, allow to process compositional forms in a generic and extensible manner.

The second and third step build upon the first step and introduce an abstraction mechanism and an implementation of the basic form components supported by HTML, such as text fields, drop down boxes and more. These steps are not within the scope of this thesis and remain to be described in future work.

\textsuperscript{10} http://www.sencha.com
\textsuperscript{11} http://dojotoolkit.org
1.3 Outline

Chapter 2 explains the concepts of web forms, crosscutting, reusability and model binding and finishes by describing the most important related work. Chapter 3 analyses the problems of form processing, both with and without abstraction layers. Chapter 4 presents a three-step approach for solving these problems and then explains the first step in detail. Chapter 5 finally sums up the content of the thesis, discusses whether the presented solution actually solves the identified problems and provides an outlook to possible future work.
2 Background

A few fundamental terms need to be explained to prepare the reader for the concepts described in this thesis. Most importantly, the reader needs to know the characteristics and technical foundation of web forms. This topic is covered in section 2.1. Section 2.2 then explains the most important syntax constructs of PHP, the language of choice in this thesis, and why this language was chosen. Section 2.3 shortly explains the crosscutting phenomenon and why it occurs in web form implementations that follow the Model-View-Controller pattern. Section 2.4 describes how reusability is important for form processing systems and introduces the reader to the terms “abstraction” and “compositional reuse”. Section 2.5 explains the concept of domain models, how forms are used to manipulate domain models and why this thesis pursues the concept of model-bound forms. Section 2.6 discusses several existing concepts that try to address these concerns. Section 2.7 finally clarifies the terminology used throughout this thesis.

2.1 Web Forms

Web forms are the web’s equivalent of paper forms. Both share common properties in that they consist of one or more labeled fields where their users are supposed to insert information. An example of a web form for editing the contents of an employee can be seen in Figure 2.1.

![Figure 2.1: A simple form for entering the personal data of an employee.](image)

While paper forms are traditionally handed over personally or via mail, web forms normally provide a button to submit the form to its server. When the user eventually clicks that button, an HTTP POST request is sent to the form’s target URL. This request contains the form’s data in simple name/value pairs, as specified by the CGI protocol. A sample request of the above form can be seen in Listing 2.1.

```
1 POST /employees/new HTTP/1.1
2 Host: example.com
3 Content-Type: application/x-www-form-urlencoded
4 Content-Length: 52
5
6 name=Jane%20Doe&salary=40000&year=1941&month=9&day=9
```

Listing 2.1: A sample HTTP POST request of `newEmployee.php`.
When a paper form arrives at its destination, its data can be validated and evaluated. But if the form turns out to contain errors, correcting these errors is difficult. It either needs to be sent back to the person who filled it out via mail, or the person needs to be phoned to ask for corrections. With web forms, the validation of user input is much easier. When a user submits a form, it is usually validated immediately. If it contains any invalid input, it is displayed again and shows error messages helping to correct the input. Figure 2.2 shows this concept.

![Figure 2.2: The form of Figure 2.1 after an invalid submission.](image)

### 2.2 PHP

Several technologies exist for processing and validating web forms on a web server. One such technology that has become very popular in the last decade is PHP\(^1\). Originally, PHP was a scripting language for creating dynamic HTML pages. Only later it developed the functional and object-oriented capabilities that it has today. PHP was chosen for this thesis because an implementation of the concept presented here already exists for the PHP framework Symfony\(^2\).

Apart from that, PHP seems like a good choice for the purposes of this thesis for three reasons:

- PHP offers a syntax similar to Java or C-based languages. It should therefore be easy to understand for a large number of developers.
- PHP is more concise than its strongly-typed relatives and leads to shorter code listings.
- PHP offers very simple interfaces to HTML generation and CGI parameters, which also contributes to concise code listings.

Nevertheless, PHP is only used to support the reader’s understanding of the points made in this thesis. The underlying concepts are intended to be generic and portable to other languages.

Listing 2.2 demonstrates a short snippet of PHP that outputs the HTML markup for the month drop-down in Figure 2.1\(^3\):

---

\(^1\) [http://php.net](http://php.net)

\(^2\) [http://symfony.com](http://symfony.com)

\(^3\) More information about the HTML tags used in this thesis can be found in the HTML 4.01 Specification [25].
Crosscutting is a phenomenon that occurs when developing web forms in a Model-View-Controller (MVC) environment [28]. The MVC paradigm is an architectural pattern separating an application into three layers:

- The **Model** layer is a simulation of the application’s business domain. It can be as simple as a set of numbers and as complex as an ontology of different inter-related classes.

- The **View** layer deals with the graphical output. It displays data from the model and listens for changes in order to update itself, usually by means of the Observer pattern [20].

- The **Controller** layer is responsible for handling user input. Based on this input, the model and the view are notified about changes.

Web forms usually touch all three layers of an MVC application. They have a visual representation displaying the various form fields to the user (the View layer). They have buttons that submit the entered data back to the server where the data is processed (the Controller layer). At last, the data manipulated by web forms is part of the Model layer. This mixture of concepts within a web form is a big problem for long-term maintenance. Consider for example that the structure of the Model layer is changed. Then also the View layer needs to be adjusted to display the corresponding form fields. At last, the Controller layer needs to be modified so that it passes the values submitted through the fields of the View back to the Model.

Kojarski and Lorenz identified this problem as **Inter-Crosscutting**. They observed that “Inter-crosscutting reveals a strong dependency of web application code on the structure concern.” They conclude that having such a structural “skeleton in the code results in a high application assembly cost, in a high development and maintenance cost, and in loss of reuse opportunities.” [27]. Crosscutting should thus be reduced to a minimum to make code development and maintenance cheaper and more efficient.
2.4 Reusability

Reuse of software can be defined as “the process of creating software systems from existing software rather than building them from scratch” [29]. Reusability then refers to the ability of a software, or a software component, for being reused. Within the context of web forms, consider for example the date field in Figure 2.1, which consists of three drop-downs to select the month, day and year of the date. The field entails display logic (the construction of the HTML code), processing logic (the transformation of the submitted values to DateTime, PHP’s data type for representing date) and validation logic (checking whether the year, month and day combination actually exists). Reuse in this case means not having to duplicate that logic when building another form with a similar (or identical) date field.

Two major techniques for software reuse are abstraction and compositional reuse [34]:

Abstraction

Abstraction is the extraction of commonly shared logic into a reusable component. As Krueger states, “Abstraction is the essential feature in any reuse technique” [29]. He continues to divide abstractions into fixed, variant and hidden parts. The fixed part of an abstraction is known to the user, but cannot be changed. The variant part is known to the user and can be adjusted. The hidden part, at last, is completely hidden away in the abstraction.

In the previous example, the fixed part for example contains the output type of the date field, which always produces a DateTime instance. The variant part could be the range of selectable dates, which needs to be configurable by the user. The hidden part refers to internal logic, such as how exactly the submitted data is converted to a DateTime instance.

Compositional Reuse

Compositional reuse, or compositionality, denotes the construction of more complex components by combining simpler components [34]. This results in a natural separation into abstraction layers. The simple components form the base layer, with ideally all other components built on top of them. Only if a component differs too much from existing components it is built from scratch.

In the previous example, the base layer of our abstraction hierarchy are the drop-down fields. These have inherent behavior and constraints and can be put into any form in order to select from a list of values. The date field is then a composition of three drop-down fields and builds specialized functionality on top of them.

As is commonly known, software reuse positively impacts the quality, reliability and performance of the resulting application while at the same time enhancing the productivity of the development team [34]. It should therefore be a first-order goal for form libraries to support abstraction and compositionality to facilitate the reuse of their components.

2.5 Model Binding

The Model layer, also called the Domain Model [22], is used to capture information about the type and structure of objects, the associations between these objects, constraints on their properties and associations as well as actions that can be executed on them. Put in other words, the Domain Model is a simulation of a part of the real world that is relevant within an application.
In object-oriented languages, the most natural way to formally describe a Domain Model is by creating classes. Consider a typical enterprise application that stores information about the employees of an organization. A simplified class `Employee` might be defined as shown in Listing 2.3.

```php
1 class Employee
2 {
3     public $name;
4     public $salary;
5     public $birthDate = new DateTime();
6 }
```

Listing 2.3: The domain model class `Employee`.

A web form for modifying such objects needs to display the actual values of the object’s properties in its fields. Once submitted, the values entered by the user need to be written back into the object, before it can be processed or persisted. Since every field of a form maps to a specific property (or set of properties), the form ideally takes care of the data exchange autonomously.

Due to its strong correlation with the domain model, a lot of information about a web form can be collected by introspecting the domain model’s metadata. For example, in strongly typed languages, the fields displayed in the form can be derived from the data types in the domain model. If a property has the type `Date`, the web form can conclude to display a date field for manipulating that property. Apart from data types, further metadata about the domain model, such as validation or persistence metadata, should be integrated into web forms. If it is already known that a property represents a one-to-many association to a different class, forms can derive the appropriate field for letting the user manipulate that association.

Finally, I define Model Binding as (I) the direct data exchange between a form and the domain model and (II) the form’s partial self-configuration by reusing the domain model’s metadata. A form that is able to fulfill these requirements is a model-bound form. One of the goals of this thesis is to find an architecture that best possibly supports this concept.

### 2.6 Related Work

Several different approaches have been taken in the last fifteen years to solve the crosscutting and reusability problems of web forms. The following developments had the biggest impact on the concepts presented in this thesis:

**Mawl**

Mawl [2, 3] is a “domain-specific language for form-based services”. Mawl makes a distinction between HTML templates and services. HTML templates are .mhtml-files that contain the HTML code and placeholders for dynamically inserted strings. Each template has a fixed number of input and output parameters. Input parameters can be passed into the template and are included in the resulting HTML. Output parameters correspond to form fields, are filled by the user and then passed back to the service. Services tie templates and the business logic of an application together. Mawl provides a domain-specific language for formulating these services with special support for sessions and forms. The big benefit of Mawl is that the services and templates can be created and maintained by different development teams. The problem of incompatibilities between the services and templates is mitigated by checking the input and output parameters used in the services against the parameters specified in the templates at compile time.
The <bigwig> project [8] is a domain specific language heavily inspired by Mawl. Like Mawl, it makes a distinction between services – formulated in a domain-specific language – and HTML templates. But while Mawl does not allow to insert HTML templates into other HTML templates, <bigwig> supports this by turning HTML into a first-class data type. An HTML template in <bigwig> is nothing more than a normal variable, except that it defines input and output parameters. The input parameters can again be HTML templates and so allow for a much more fine-grained reuse of HTML snippets than Mawl. Like in Mawl, the assignment of input and output parameters is checked at compile time.

iData

The iData Toolkit [30, 31, 32] is a form library for the functional programming language Clean. iData encapsulates form fields within iData elements. Each iData element has a state and a bijective function for converting that state to its view representation and back. Due to this conversion function and Clean’s strongly-typed nature, iData is safe from passing invalid values to the application. iData elements can also be interconnected. For example, when one iData element is changed, the constraints of another iData element can be changed accordingly. Last, iData elements are compositional. Higher-level elements can be created by composing lower-level elements together. One example brought in [31] is a counter input with a textual input for numbers, a button to increase and another to decrease the number by one.

Django

Django4 is a full-stack web framework for the object-oriented, interpreted language Python. It features a convenient library for processing web forms, where fields are represented by instances of the class Field. This class contains the conversion logic between the model’s and the view’s data types and the logic for enforcing validation constraints. Each field also contains a Widget instance, which creates the HTML representation for that field. Forms are instances of the class Form and contain one or more fields, but cannot contain nested forms. Django features a special class ModelForm that generates forms for ActiveRecord instances by introspecting their metadata. Django does not support the composition of fields to more complex fields.

WUI

The Web User Interface (WUI) library [23, 24] is a form library for the declarative multi-paradigm language Curry. A WUI element is a building block of a form, such as a text input or a drop-down box. WUIs are “type-oriented”, that means that each WUI element only accepts model values of specific types and also guarantees to output values of that type. WUI elements can also be composed using WUI combinators, to manipulate tuples or custom complex data types. Validation constraints can be defined for each WUI element to restrict the domain of its submitted values. A form, at last, is a combination of a – in most cases composed – WUI element, its initial data and a handler function that is executed once the form is submitted. The HTML rendering of WUIs is decoupled and can be overridden by providing a renderer function.

Formlets

Formlets [12, 14] are abstracted, compositional form elements for the programming language OCaml. This library is inspired by iData and WUI, but tries to ”reduce form abstraction to its essence“ by means of idioms. It is similar to its predecessors in that a

---

4 https://www.djangoproject.com

An algorithm and architecture for server-side processing of model-bound compositional web forms
**formlet** is an element of a form that collects rendering, data transformation and validation logic and can be easily reused. Formlets can be composed to create higher-level formlets that output complex data types. Basic implementations of formlets have been created for Haskell, OCaml, Python and C# [12].

For the simplification of further analysis, it is helpful to categorize the abstraction strategies of these approaches into two groups:

**HTML Abstraction**

The first group abstracts at the level of HTML code. By pushing HTML code into separate templates with placeholders that can be filled with variables, the HTML code is made reusable. Concepts like Mawl, <bigwig> and JWIG [11] belong to this group.

**Form and Field Abstraction**

The second group abstracts at the level of forms and their fields. Fields, their values, their HTML representation, their type conversion code and their validation constraints are combined into abstract, reusable components. These components can often be composed in order to form higher-level fields or forms. WASH [35], iData, Ruby on Rails\(^5\), Django, WUI and Formlets belong to this group.

### 2.7 Terminology

In the course of this thesis, we will need a terminology for distinguishing between abstract and concrete fields. Very much like classes in object-oriented programming, abstract fields are blueprints for creating concrete fields. I will refer to them as **field types**. Concrete fields are instances of field types as they occur in specific forms. I will refer to them as simply **fields**. For example, consider a form that allows to enter information about a famous author. The form contains two **fields**, one for entering the birth date and another for entering the death date. While these fields are logically different, they incorporate the same display and processing logic, which is inherited from their common **field type**, namely "date".

\(^5\) [http://rubyonrails.org](http://rubyonrails.org)
3 Problem Specification

In order to develop a concept that addresses both the solved problems and the shortcomings of existing form libraries, these problems and shortcomings need to be specified. As a foundation for the following analysis, section 3.1 starts with a simple but typical, abstraction-free example of a web form and its processing code. Section 3.2 lists the generic problems that form libraries have to handle and relates them to the example. The existing libraries address these problems in different ways and depending on their approach, they face various new problems. Section 3.3 explores the challenges faced by form libraries abstracting at the level of HTML code. Section 3.4 finally discusses the problems that must be solved by libraries abstracting forms and fields into reusable components (see section 2.6).

3.1 Introductory Example

We start by looking at a typical example of a web form in PHP. As concise as the example may be, it will include all the typical steps of form processing and thus will be sufficient for illustrating the common problems that can be found there. These steps are:

1. Read default values from the domain model and convert them to strings.
2. Display the form with its default values.
3. Handle the form submission.
   a) Validate the submitted values.
   b) If the form is invalid, show it again and display error messages at the invalid fields. The fields should now contain the submitted instead of the default values.
   c) If the form is valid, convert the submitted values back to the model types and write them into the domain model.

This example is intentionally written in a procedural style without using any other abstraction layers over PHP, except for a class Request that facilitates the access to the CGI parameters. It is intended to highlight the problems of form processing as independent of abstraction layers, frameworks and programming languages as possible.

Listing 3.1 shows the source code of the form shown in Figure 2.1. The code will be stored in a file newEmployee.php and corresponds to the View layer of our sample application.

```php
<form action="/employees/new" method="POST">
  <div>
    <label>Name</label>
    <input type="text" name="name" value="<?= $nameVal ?>">
    <? if (isset($errors["name"])): ?>
      <span class="error"><?= $errors["name"] ?></span>
    <? endif ?>
  </div>
  <div>
    <label>Salary</label>
    <input type="text" name="salary" value="">
    <? if (isset($errors["salary"])): ?>
      <span class="error"><?= $errors["salary"] ?></span>
    <? endif ?>
  </div>
</form>
```

An algorithm and architecture for server-side processing of model-bound compositional web forms
Chapter 3. Problem Specification 3.1. Introductory Example

Listing 3.1: The template newEmployee.php.

The Controller layer takes care of processing the submitted request. Listing 3.2 shows sample PHP code for such a controller. The Employee class is the same as in Listing 2.3. The $request variable is assumed to contain an instance of the previously mentioned class Request, which exposes various details of the HTTP request in a convenient fashion. The method `getMethod()` returns the HTTP method of the request, such as “GET”, “POST” or “HEAD”. The method `getParameter()` returns the CGI parameter with the given name.

```php
$employee = new Employee();
$nameVal = $employee->name;
$salaryVal = (string) $employee->salary;
$yearVal = $employee->birthDate->format("Y");
$monthVal = $employee->birthDate->format("m");
$dayVal = $employee->birthDate->format("d");
if ("POST" === $request->getMethod()) {
    $nameVal = $request->getParameter("name");
    $salaryVal = $request->getParameter("salary");
    $yearVal = $request->getParameter("year");
    $monthVal = $request->getParameter("month");
    $dayVal = $request->getParameter("day");
}
$nameVal = trim($nameVal);
$salaryVal = trim($salaryVal);
```

An algorithm and architecture for server-side processing of model-bound compositional web forms 12/46
$errors = array();

if ("" === $nameVal) {
    $errors["name"] = "Please enter a name";
}

if ("" === $salaryVal) {
    $errors["salary"] = "Please enter a salary";
} elseif (is_numeric($salaryVal)) {
    $salary = (float) $salaryVal;
} else {
    $errors["salary"] = "Please enter a valid number";
}

try {
    $birthDate = new DateTime("$dayVal/$monthVal/$yearVal");
    if ($birthDate >= new DateTime("today")) {
        $errors["birthDate"] = "The date must lie in the past";
    }
} catch (Exception $e) {
    $errors["birthDate"] = "Please select a valid date";
}

if (0 === count($errors)) {
    $employee->name = $nameVal;
    $employee->salary = $salary;
    $employee->birthDate = $birthDate;
    /* persist $employee and redirect to a success page */
}

/* render newEmployee.php */

Listing 3.2: The controller for processing the form in listing 3.1.

In line 1, a new domain model instance is created. Real applications would load the model from secondary storage, as defined by the application’s need to modify existing, persisted data. Lines 3–7 convert the properties of the domain model into strings that are later passed to the View layer. Line 9 checks whether the form has been submitted. If this is the case, the code continues to override the view variables with the submitted request parameters in lines 10–14. These lines ensure that the submitted values are preserved in the form when the validation fails and the form is displayed again. Lines 16–17 remove leading and trailing whitespace from the submitted name and salary. Lines 19–41 validate that the submitted values match the constraints of the application. In Line 28 and 34, the strings sent in the HTTP request are converted back to the appropriate PHP types and validation continues. If the validation succeeds, lines 44–46 update the model. Finally, the model is persisted to secondary storage and the browser is redirected to a page indicating that the form was processed successfully.

### 3.2 Generic Problems

Several generic problems can be identified in Listing 3.1 and 3.2:
Name Mismatch

The names of the form fields defined in the View (Listing 3.1, lines 4, 11, 18, 28 and 31) must match the names of the request parameters accessed in the controller (Listing 3.2, lines 10–14). If a name is changed in only one of the layers, a runtime error occurs. This problem is a crosscutting problem as described in section 2.3.

Impedance Mismatch

The programmer must manually convert the application data into strings (Listing 3.2, lines 3–7) and the request parameters back to the application data types (Listing 3.2, lines 28 and 34). This is called the impedance mismatch problem [13]. Since HTTP requests are text based, while PHP supports a rich variety of data types, the data handled by forms must be converted back and forth between these different representations.

Dirty Data

If the programmer wants to clean submitted values from unexpected input, such as automatically removing leading and trailing spaces to protect end users from typing mistakes, he needs to do so manually (Listing 3.2, lines 16–17). This leads to unnecessary duplication of code if applied in the same fashion to multiple fields or in multiple forms.

Invalid Data

Submitted parameters must be validated against constraints required by the application domain (Listing 3.2, lines 19–41) and error messages must be displayed next to the invalid fields (Listing 3.1, lines 5–7, 12–14 and 34–36). The responsible code must be programmed explicitly, leading to a waste of development time and annoyed developers [9].

Validation Mismatch

If additional validation constraints are defined in the Model layer, these need to be synchronized with the validation performed in the controller. Listing 3.3 shows the rewritten Employee class using JSR 303-like [5] constraint annotations found in the Symfony2 Validator

```
class Employee
{
    /** NotNull MinLength(3) */
    public $name;

    /** NotNull Past */
    public $birthdate = new DateTime();
}
```

Listing 3.3: The Employee class with constrained properties.

If the controller does not enforce a name to be present in the form (Listing 3.2, lines 21–23), the validation of the Employee object will fail if invoked at a later point, for example by the storage mechanism. This will lead to a runtime error instead of showing an error message to the user that helps him to correct the form.

Code Duplication

If the same field exists several times within a form, all its associated HTML code and processing logic must be duplicated. For example, consider that a new property $registrationDate is added to the Employee class. Like the $birthdate property, this

---

1 https://github.com/symfony/Validator
property should be edited by a date field. Thus, all the code related to the display (Listing 3.1, lines 16–37) and processing (Listing 3.2, lines 5–7, 12–14, 33–41 and 46) of the $birthDate property must be duplicated and adapted.

Fowler et al. define code duplication as the worst code smell [19]. Duplicated code increases the code size, indicates bad design and generally complicates maintenance, because bug fixes or changes need to be applied manually to all copies of the duplicated code [16].

Cross-Site Request Forgery

A Cross-Site Request Forgery (CSRF) attack\(^\text{2}\) is a security attack that takes actions on behalf of an unknowing user. For example, consider a banking application with a form for transferring money. Consider next an attacker with a malicious website that contains a form with the same structure and target URL as the banking form, except that all its fields are hidden and prefilled, with the intent of transferring money from your to the attacker’s account. Whenever the form is submitted, its data will be handled by the banking application as if it was submitted through its own form. If the attacker can trick a user of the banking application to submit while his banking session is active, the banking application won’t be able to distinguish between the fake and an authentic request and process the transaction. Different defense mechanisms exist [4, 26] and should be integrated into form libraries to mitigate this threat.

3.3 HTML Abstraction Problems

One group of form libraries tries to solve the Code Duplication problem by extracting the HTML code of form fields into separate, reusable files (see section 2.6). As an example, consider that the HTML code for rendering date fields is extracted (Listing 3.1, lines 16–37). It can then be reused whenever a date field should be rendered. While this solution avoids writing duplicate HTML code, it provokes two problems that did not exist before:

**Name Clashes**

If the field names in the reusable HTML template are static (for example date_year, date_month and date_day), they will collide if the template is included more than once. As shown before, the CGI protocol requests form data to be sent in name/value pairs. Every name can only exist once in the request.

This problem is also apparent in forms that don’t use abstraction layers, where the field names are assigned manually. However, since the programmer consciously has to chose each name there, he is less likely to use the same name twice.

**Structural Duplication**

Structural changes in the reusable HTML template require the programmer to apply the same structural changes to the controller. For example, consider that the date field is changed to be represented by a single text input field instead of the three select boxes. Such a change requires the programmer to change the type conversion code (Listing 3.2, lines 5–7, 28 and 34) in each controller that uses the field. This problem belongs to the group of crosscutting problems as described in section 2.3.

\(^\text{2}\) http://www.tux.org/~peterw/csrf.txt
3.4 Field Abstraction Problems

The second group of form libraries abstracts forms and fields into reusable field types (see section 2.6) that encapsulate not only their HTML code, but also the code responsible for type conversion (Listing 3.2, lines 5–7, 28 and 34) and validation (Listing 3.2, lines 19–41). The field types are usually programmed against specific data types and know how to convert these types to plain text – or tuples of plain text, if their HTML representation contains more than one field – and back. But also in this group of libraries we can identify new problems that need to be addressed:

Type Mismatch

The data types of the domain model must always match the data types of the associated fields. If the data type of a property is changed, a different field type has to be used. Like Name Mismatch, this problem is a crosscutting problem as described in section 2.3.

Model Update

Even though the fields take care of converting the user input to typed values, their actual output must be assigned manually to the properties of the domain model (Listing 3.2, lines 44–46). When properties are added to or removed from the domain model, changes must be cascaded here accordingly. This problem also is a crosscutting problem as described in section 2.3.

Prepopulation

Similarly to Model Update, the default values of the fields need to be set manually by reading the properties of the domain model (Listing 3.1, line 4, 11, 20, 23 etc.). Properties cannot be added to or removed from the domain model without cascading the corresponding assignments of default values.

Specialization

Field types should be specializable. For example, one might repeatedly use date fields for entering birth dates. Instead of duplicating all the birth date logic, a much more elegant way is to specialize the date field into a new, reusable field type that only shows years in the past.

In object-oriented programming languages, the most natural specialization mechanism is inheritance. Specialization by inheritance can be divided into horizontal extension and vertical extension [36]. Horizontal extension is the addition of functionality to a component. Vertical extension, on the other hand, is the restriction of a component to a subset of its functionality. Field types should support both kinds of extension.

Mixins

It should be possible to attach custom functionality to existing field types. For example, the developer might find it necessary to add an asterisk (“*”) to the labels of all required fields. Solving this problem by using specialization would mean to create a specialization of every existing field type, which again leads to duplication of code. A more maintainable solution would be to develop the functionality once and attach it to an existing field type and all of its heirs in the inheritance tree. This kind of extension is called Mixin [10].

Redefinition

Field types should be completely exchangeable. For example, consider replacing the default “date” field type by a custom implementation that allows to manipulate the date
through a single, localized text input instead of three dropdown elements as in the default implementation. The most obvious solution is to create a custom field type, such as “localized_date”, and to use that type instead of the “date” type. The drawback of this solution is that it will not affect types extending from the “date” type, for example “birth_date”. Thus, a more appropriate solution would be to redefine the “date” type and in this way completely replace it by the custom implementation.

**Compositionality**

Fields have to be composable. A simple example for this requirement is a date field that consists of three dropdown fields for choosing the year, month and day of the date. If the date field cannot compose the dropdown fields, it needs to completely duplicate their logic internally. Forms should also be composable. For example, consider a form used to change the password of a user. Another form for creating a new user account will probably also have the requirement to let the user choose a new password. Embedding the password form into the signup form will prevent the developer from unnecessarily duplicating the password form logic.

**Dynamic Forms**

Today, many web applications employ JavaScript to make form input more dynamic. With JavaScript, fields can implicitly be shown, hidden or changed based on the input that the user provides. A common use case can be found in typical address forms. Users first select their country and, based on their selection, a second field is shown that allows to choose among the provinces or states of that country. Other typical forms allow users to explicitely add, remove or change fields. In profile forms, for example, users can often enter a variable number of email addresses by clicking “Plus” and “Minus” symbols that add new or remove existing email fields.

While JavaScript provides great facilities to make this dynamic aspect of forms accessible to the user, it also makes form processing more difficult. The web server can no longer render a form as HTML and expect the exact same form to be submitted back. Instead, it needs to adapt its internal form model to match the submitted CGI parameters. It also needs to replicate the logic of to the JavaScript layer to validate whether the submitted CGI parameters actually correspond to that logic, because we cannot guarantee that an incoming HTTP request was really generated by our application.
4 Concept and Implementation

This chapter presents an approach for solving the problems described in chapter 3. Section 4.1 outlines a step-by-step approach to that solution comprising three different steps. The rest of the chapter describes the first of these steps in detail. Section 4.2 presents a generic object model for web forms. Section 4.3 explains a generic solution for transforming data types of the domain model to the types used in the HTML output and back. Section 4.4 describes a strategy for naming form fields and binding CGI parameters into the form model. Section 4.5 analyzes how to map fields in the form model to objects and properties of the domain model. Section 4.6 explains how events help to deal with the dynamic aspects of form processing. Section 4.7, at last, summarizes the algorithm that ties the concepts presented in this chapter together.

4.1 Three-Step Approach

In order to make the solution to the problems described in chapter 3 more tangible, it is helpful to work with a step-by-step approach, where each step solves a subset of these problems and acts as foundation for the following steps. For the concept presented in this thesis, I chose a separation into three steps that are described below, together with the problems they solve.

1. Generic Form Model and Algorithm Design

In the first step, a generic form model is designed. This model must be capable of representing arbitrary constellations of forms and fields (Compositionality). It must support an algorithm that is able to extract default values from the domain model (Prepopulation), convert them to strings for use in the HTML output (Impedance Mismatch), provide a naming strategy for the HTML fields that is later used to bind the submitted values back to the form model (Name Mismatch, Name Clashes), clean and validate the submitted values (Dirty Data, Invalid Data) and write them back into the domain model (Model Update). This algorithm must be configurable to match the requirements of specific forms. The prepopulation and especially the binding process must support self-adaptation of the form model to its default values and the values submitted by the user (Dynamic Forms).

2. Generic Meta-Model Design

The next step is to make specific configurations of form models and algorithms reusable (to avoid Code Duplication and Structural Duplication). We do so by designing a meta-model that acts as a blueprint for these configurations, much like classes in object-oriented design are blueprints for their instances. The entities of this meta-model are form types, for example “author” for an author form, “article” for an article form or “text”, “date” and “email” for basic fields. The meta-model must support Specialization, Mixins and Redefinition of form types. Furthermore, instances of the model must be configurable in order to match the specific requirements of their environment. For example, when creating an instance of the “text” type, it should be possible to set the allowed maximum length of the entered text. At last, the meta-model should support the derivation of configuration from existing metadata in the domain model (to avoid Type Mismatch and Validation Mismatch). For example, if the domain model contains a property that is marked as non-empty email address, the meta-model should reuse this information when a field that is mapped to that property is created.
3. HTML-Specific Meta-Model Implementation

The last step is to implement the meta-model with specific, HTML-specific form types. This step includes more subtle challenges such as finding an appropriate form type ontology, preventing CSRF-attacks and solving the intrinsic problems of each individual form type, such as localized date and number input or the population of drop down fields from the domain model.

The remainder of this thesis focuses on the first step, namely designing a generic form model and processing algorithm. The detailed description of the second and third step is subject to future work.

4.2 Form Model

A convenient way for determining a model for a domain is to create an object for each of the domain’s nouns [17]. We will do so by creating a model for the example given in section 2.1. At first, we can identify the HTML input fields, which we will call simple fields in the model. Next, we have a form which consists of a set of such fields. At last, we can group the fields “year”, “month” and “day” to a type that yields a DateTime instance. We will call this type a compound field. Figure 4.1 illustrates this model.

As we can see in the figure, all of the objects in the model contain data that is displayed in and modified by them. For the form, the modified data is an Employee instance. For the compound field, the modified data is the DateTime instance of the employee. The simple fields, finally, contain unstructured values that are displayed and edited directly in the view: the name and salary of the employee and the year, month and day of the birth date.

The simple fields furthermore store a name that is rendered in the HTML code. Since this name must be unique, it can also be used as identifier of the field within its surrounding form. Consequently, all fields in the form should have an identifier, so we need to add a name property to compound fields as well.

This model is however still susceptible to the Compositionality problem. The children of Form are assumed to be instances of either SimpleField or CompoundField. It is not possible to nest forms within forms. To solve this problem, the three classes Form, SimpleField and

---

Figure 4.1: An intuitive model for the example in section 2.1.
CompoundField are reduced to a single class, which implements the Composite pattern \[20\] instead. Here, this class will simply be called Form. Figure 4.2 illustrates this approach.

The demonstrated design is similar to the design of WASH, iData, WUI and Formlets (see section 2.6). All of these libraries model forms and their elements as instances of the same type or, at least, as instances of types extending the same base type.

Finally, an interface Form can be derived from the classes shown in the model. This interface is shown in Figure 4.3.

The interface supports the following methods:

- **getName**: Returns the name of the form.
- **getData**: Returns the data of the form.
- **setData**: Prepopulates the form with given data.
- **bind**: Updates a form with data from a form submission.
- **add**: Adds a child to the form.
- **get**: Returns the child with the given name.
- **remove**: Removes the child with the given name.
Chapter 4. Concept and Implementation

4.3. Data Transformation

<table>
<thead>
<tr>
<th>Model Domain (M)</th>
<th>View Domain (V)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M_{\text{Dollars}} : \text{float} )</td>
<td>( V_{\text{EnglishNumber}} : \text{string} )</td>
</tr>
</tbody>
</table>

Table 4.1: The domains of the salary field.

The interface also needs to support several other methods that will be described in the following sections of this chapter.

Classes implementing this interface must not depend on their parent or their children. Instead, they should encapsulate their own responsibility and delegate everything else to the responsible instances. For example, bind should update a form’s state, but delegate the state update of its children to these by forwarding the bind call. The same applies for setData. Sections 4.4 and 4.5 explain these methods in more detail.

4.3 Data Transformation

The model presented in section 4.2 has a severe limitation: It does not offer a solution to the Impedance Mismatch problem. Data is stored in the representation used in the domain model, but not in the representation used in the HTML source code.

iData and WUI approach this issue by combining each field with a bijective transformation function \( t : M \rightarrow V \). This function transforms values from the model domain \( M \) to the view domain \( V \), which typically contains only strings. The salary of the employee, for example, is stored as a float \( m \in M_{\text{Dollars}} \). \( t \) converts the value to a string \( v \in V_{\text{EnglishNumber}} \) with an English comma delimiter (i.e. a dot) and a fixed number of decimals (see Table 4.1). This string is displayed to and modified by the user. The modified string \( v' \) is then submitted back to the server and transformed back to \( m' \in M_{\text{Dollars}} \) by applying \( t^{-1} \).

\[
v = t(m) \\
m' = t^{-1}(v')
\]

It should be obvious why we are talking about “domains” and not “data types” here. The view domain in the previous example was \( V_{\text{EnglishNumber}} \) of type string. We could change that domain to \( V_{\text{GermanNumber}} \) – a number with a German comma delimiter – and would end up with an entirely different view domain, that still has the type string.

For our object-oriented form model, we will wrap \( t \) into a class so that it can easily be swapped with other implementations. Such classes already exist in other frameworks, for example Value Transformers in Objective-C\(^1\) or Converters in JavaServer Faces (JSF)\(^2\). These are simple classes with two methods corresponding to \( t \) and \( t^{-1} \). We will choose the same approach and create a new interface DataTransformer that is shown in Figure 4.4.

![Interface](image)

Figure 4.4: The interface for transforming data between \( M \) and \( V \).

---


\(^2\) [http://docs.oracle.com/cd/E17802_01/j2ee/javaserverfaces/1.2_MR1/docs/api/javax/faces/convert/Converter.html](http://docs.oracle.com/cd/E17802_01/j2ee/javaserverfaces/1.2_MR1/docs/api/javax/faces/convert/Converter.html)

An algorithm and architecture for server-side processing of model-bound compositional web forms 21/46
The interface supports the method `transform`, corresponding to \( t \), and its bijective counterpart `reverseTransform`, that corresponds to \( t^{-1} \). Figure 4.5 illustrates the transformation process from \( M \) to \( V \). The process is executed when `setData` is invoked on a form in order to populate it with default data. The data is passed to the associated transformer and transformed to a string which can be printed in the HTML output. The thick border in the figure indicates the element whose state is changed throughout the process.

![Figure 4.5: Transformation from \( M \) to \( V \) when invoking `setData`.](image)

Figure 4.5: Transformation from \( M \) to \( V \) when invoking `setData`.

Figure 4.6 illustrates the reverse transformation process. This process is started when `bind` is called on a form in order to update it with a submitted value. The submitted string is passed to the `reverseTransform` method of the transformer to convert it to a value \( m \in M \).

![Figure 4.6: Reverse transformation from \( V \) to \( M \) when invoking `bind`.](image)

Figure 4.6: Reverse transformation from \( V \) to \( M \) when invoking `bind`.

If the reverse transformation is not possible (for example, because the user made a mistake and the given string does not conform to the rules of the transformer), `null` should be stored as model data of the form instead. However, the form must also store the erroneous view data in order to let the user correct the value. Because the model data and the view data now do not contain the same content anymore, the form will be marked as not synchronized. Unsynchronized forms must be ignored during the validation; validating \( m \) does not make sense if the transformation from \( V \) to \( M \) failed and potentially causes false positive errors.

Two new methods must be added to the interface `Form`. Furthermore, the description of `getData` needs to be refined:

- **getDataSet**: Returns the data of the form in the model domain \( M \).
- **getViewData**: Returns the data of the form in the view domain \( V \).
- **isSynchronized**: Returns whether the data in the different domains contains the same content.
Transformation Chains

Unlike JSF’s Converters, which restrict $V$ to string values, $M$ and $V$ in this implementation may contain values of arbitrary types. This allows chaining of data transformers and enables developers to extend existing forms for new input or output types.

For example, consider a predefined “money” field like in Table 4.1 which converts dollars to English numbers by means of a transformer $t : M_{Dollars} \rightarrow V_{EnglishNumber}$ as described above. If the developer wants his application to process money in cents instead, he needs to be able to change the model domain $M$. An intuitive approach to this problem is the introduction of transformation chains. By default, the transformation chain $c : M_{Dollars} \rightarrow V_{EnglishNumber}$ will only contain the original transformer $t$. The developer can then create a new transformer $u : M_{Cents} \rightarrow M_{Dollars}$ and add it to the beginning of $c$, which then becomes $c' : M_{Cents} \rightarrow V_{EnglishNumber}$. Whenever `setData` is called with some data $m$, all transformers in the chain are executed in order:

$$v = c'(m) = (u \circ t)(m) = t(u(m))$$

When `bind` is invoked with a submitted value $v'$, the reverse transformation methods are executed in reverse order:

$$m' = c'^{-1}(v') = (t^{-1} \circ u^{-1})(v') = u^{-1}(t^{-1}(v'))$$

With the customized transformation chain, the money field now returns integers to the application without duplicating the logic in the original transformer $t$. The domains of the customized field are summarized in Table 4.2.

<table>
<thead>
<tr>
<th>Model Domain ($M$)</th>
<th>View Domain ($V$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_{Cents} : integer$</td>
<td>$V_{EnglishNumber} : string$</td>
</tr>
</tbody>
</table>

Table 4.2: The domains of the customized money field.

Normalization

The money field described in the previous sections is still fairly limited. It can only handle the domains $M_{Dollars}$ and $V_{EnglishNumber}$. Any customization must be done by creating custom transformers and inserting them into the transformation chain as described above.

This is not an ideal situation. What if you frequently use money fields with the model domain $M_{Cents}$ or the view domain $V_{GermanNumber}$? What if all these different variations use the same HTML rendering and internal validation logic, for example, that the submitted value cannot have more than two decimals? It clearly does not make sense to split the implementation into separate field types here.

A better solution would be to create a generic money field type that provides implementations for multiple model and view domains which can be freely combined. Whenever a field is instantiated, concrete domains $M$ and $V$ are passed and are used to configure the transformers of the field. Table 4.3 depicts this idea.

While great in theory, this idea however sparks two problems:

- The number of required transformers grows polynomially in $|M| \cdot |V|$.
- Internal logic of the field, such as validating that the number has no more than two decimals, cannot be programmed against a fixed domain anymore.
These problems can be solved by introducing a fixed intermediary domain, the normalized domain $N$. Instead of directly transforming from $M$ to $V$, we first transform from $M$ to $N$ and then from $N$ to $V$. Consequently, we only have to implement $|M| + |V|$ transformers. Internal logic, like the validation mentioned before, can be programmed directly against $N$.

For the example of the money field, a possible choice for $N$ is $N_{Dollars}$. The result is illustrated in Table 4.4.

<table>
<thead>
<tr>
<th>Model Domain ($M$)</th>
<th>Normalized Domain ($N$)</th>
<th>View Domain ($V$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_{Dollars}: float$</td>
<td>$N_{Dollars}: float$</td>
<td>$V_{EnglishNumber}: string$</td>
</tr>
<tr>
<td>$M_{Cents}: integer$</td>
<td></td>
<td>$V_{GermanNumber}: string$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$V_{FrenchNumber}: string$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$V_{ItalianNumber}: string$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$V_{SpanishNumber}: string$</td>
</tr>
</tbody>
</table>

Table 4.3: A money field with configurable domains.

Table 4.4: The money field with the additional normalized domain.

Formally, we need to split the transformation chain $c : N \rightarrow V$ into two chains $c_M : M \rightarrow N$ and $c_V : N \rightarrow V$. For example, when a money field is instantiated for the model domain $M_{Cents}$ and the view domain $V_{FrenchNumber}$, then $c_M = t$ where $t : M_{Cents} \rightarrow N_{Dollars}$, and $c_V = u$ where $u : N_{Dollars} \rightarrow V_{FrenchNumber}$. In the case that the money field is instantiated for the model domain $M_{Dollars}$, we end up with the same model domain and normalized domain $M_{Dollars} \equiv N_{Dollars}$. So we can simply put $c_M = id$, where $id$ is the identity function $id(x) = x$ for all $x$.

Whenever `setData` is invoked with some $m \in M$, $m$ is first transformed to $n \in N$ by passing it to all transformers in $c_M$. Subsequently, $n$ is passed to all transformers in $c_V$ to obtain $v \in V$:

$$n = c_N(m) = (t_{N0} \circ t_{N1} \circ \ldots \circ t_{Nk})(m)$$

$$v = c_V(n) = (t_{V0} \circ t_{V1} \circ \ldots \circ t_{Vl})(n)$$

Likewise, when `bind` is invoked with an updated value $v' \in V$, $v'$ is first passed to all transformers in $c_V$, this time in reverse order, to obtain $n' \in N$. $n'$ is then passed to all transformers in $c_M$, again in reverse order, to calculate the updated value $m' \in M$:

$$n' = c_V^{-1}(v') = (t_{Vl}^{-1} \circ t_{Vk-1}^{-1} \circ \ldots \circ t_{V0}^{-1})(v')$$

$$m' = c_N^{-1}(n') = (t_{Nk}^{-1} \circ t_{Nk-1}^{-1} \circ \ldots \circ t_{N0}^{-1})(n')$$

Forms store their data in all three domains. The interface `Form` already has two methods `getData()` and `getViewData()` for retrieving the data in $M$ and $V$. A last method needs to be added for accessing the data in $N$:

- `getNormData`: Returns the data of the form in the normalized domain $N$. 

An algorithm and architecture for server-side processing of model-bound compositional web forms
4.4 Form Naming and Binding

As mentioned before, the CGI protocol demands that the values submitted through a form be sent as list of simple name-value pairs. Listing 4.1 shows an example list for the form given in section 2.1.

```plaintext
1  name=Jane Doe
2  salary=40000.00
3  year=1941
4  month=9
5  day=9
```

Listing 4.1: Example CGI parameters sent by the form in section 2.1.

The names in this list equal the “name” attributes of the HTML fields in the form. Every of these fields is represented by exactly one leaf of the form tree and has the same name as this leaf.

In order to update the data of these leaves, the controller must pass the CGI parameters to the topmost form in the tree (the root form) by invoking its `bind` method. This method must forward the parameters to the form’s children, until eventually the leaves are reached. These, finally, read the CGI parameter with the corresponding name and update their data with the submitted value. Since the names of the form elements are now stored only once, namely within the `Form` instances, the Name Mismatch problem is solved.

Figure 4.7 shows an illustration of this approach. Thick borders mark the objects whose state is updated during binding.

Figure 4.7: Data distribution during binding.

This approach, however, does not solve the Name Clashes problem. If we duplicate the date field in order to submit another value of type `DateTime`, the names of the nested simple fields “year”, “month” and “day” will appear twice in the form. Consequently, the values sent by either of the date fields will be lost.

A solution to this problem is to make a distinction between a field’s name and its HTML name:

- The `name` of a field is used to identify this field within its immediate parent. No form may contain two immediate children with the same name. For example, it is illegal to add another field with the name “birthDate” to the employee form in Figure 4.7. It is, however,
possible to add a field named “year”, because the employee form does not contain an immediate child with that name.

• The HTML name of a field is displayed in the HTML markup, sent in the HTTP request and used to bind submitted values to their originating fields. Since the CGI protocol requires data to be sent in non-recursive lists of name-value pairs, each HTML name must be unique within the whole form tree.

One approach for constructing the HTML name is to concatenate the names of a field and of all of its ancestors in the form tree. Formally, this can be expressed in two rules:

1. If a form has no parent or a parent with an empty HTML name, the HTML name is identical to the form’s name.
2. Otherwise the HTML name is set to the parent’s HTML name with the form’s own name appended in squared brackets.

Listing 4.2 shows an example CGI request when following these rules.

```bash
1 employee[ name ] = Jane Doe
2 employee[ salary ] = 40000.00
3 employee[ birthDate ][ year ] = 1941
4 employee[ birthDate ][ month ] = 9
5 employee[ birthDate ][ day ] = 9
```

Listing 4.2: The CGI parameters generated by the new naming scheme.

This naming scheme plays well with PHP, because PHP automatically transforms names of this format into nested, associative arrays. Listing 4.3 shows the array generated by PHP when the CGI parameters shown in Listing 4.2 are received.

```php
1 array
2   "employee" => array
3       "name" => "Jane Doe",
4       "salary" => "40000.00",
5       "birthDate" => array
6         "year" => "1941",
7         "month" => "9",
8         "day" => "9"
9 }
10 }
11 }
```

Listing 4.3: PHP’s interpretation of the parameters in Listing 4.2.

Data submitted this way furthermore greatly simplifies the binding process, since it is not necessary anymore to pass the full list of submitted values to all of the forms within the tree. Instead, forms can pass only the relevant parameter subsets to their children. Figure 4.8 illustrates this idea. Again, thick borders mark the elements whose state is changed throughout the process.

The controller passes the content of the “employee” entry in Listing 4.3 to the root form. The root form then forwards the content of the “name” entry to the name field and the content of the “birthDate” entry to the date field. The latter distributes the remaining content in the same manner. The leaves of the form, at last, are bound with simple, scalar values that replace the data stored formerly.
4.5 Data Mapping

The definition of the `bind` method in section 4.2 states that `bind` “updates a form with data from a form submission”. The previous section explained how the state of the leaves in the form tree is updated. The state of the inner nodes, however, has not been changed so far. In order to complete the binding process for the inner nodes, their data must be updated with the reverse-transformed data fetched from the leaves.

Figure 4.9 illustrates this idea. The employee form contains an `Employee` object as data, which is depicted in the figure by the variable `$empl`. The object’s properties are updated by calling `getData` on each of the form’s children after binding them. The results are then passed to the corresponding setter methods `setName`, `setSalary` and `setBirthDate`. The controller, finally, can obtain the updated `Employee` object by calling `getData` on the root form.

Likewise, after passing an `Employee` object `$empl` from the controller to the form, `$empl`’s data must be distributed to the form’s children after binding them. This process is depicted in Figure 4.10. First, `setData` is called with `$empl` on the employee form from the controller. Then, the methods `getName`, `getSalary` and `getBirthDate` are invoked on `$empl`. The results of these methods are forwarded to the children of the form by calling the appropriate `setData` methods.

As can easily be spotted, this process, which I call data mapping, is symmetrical. It consists of two functions $m_f$ and $m_d$. The first, $m_f$, maps some data structure $d$ onto a list of forms $[f_0, f_1, \ldots, f_k]$ and updates the state of these forms by invoking the `setData` method.
second, $m_d$, maps the list of forms back by invoking the `getData` methods and by writing the results back into $d$, thereby modifying the state of $d$.

$$m_f : (d, [f_0, f_1, \ldots, f_k]) \rightarrow [f'_0, f'_1, \ldots, f'_k]$$

$$m_d : (d, [f_0, f_1, \ldots, f_k]) \rightarrow d'$$

In the object-oriented model, these functions are represented by the interface `DataMapper` illustrated in Figure 4.11. The interface supports two methods, `mapDataToForms` and `mapFormsToData`, fulfilling the purpose of $m_f$ and $m_d$.

![Figure 4.10: Data distribution after setting the default data.](image)

![Figure 4.11: The interface for mapping a form’s data onto its children and back.](image)

### Mapping Approaches

The argument `data` in all of these methods is a structured value which can be read and written by the specific implementation of `DataMapper`. The default implementation shipped with Symfony2, for example, reads and writes both associative arrays and objects. If an array is passed, the keys of that array are assumed to be identical to the names of the form’s children. If an object is passed, it is supposed to have either public properties with the same names as the children, or public setter and getter methods with these names.

Furthermore, Symfony2’s implementation allows to decouple the form model from the domain model. A form’s child can be named “name”, but access the “firstName” property of the data object instead. It might also be configured to access a property of a nested object or an object nested within a nested object. In short, every child of the form can be mapped to any property reachable from the form’s data object. This concept is schematically illustrated in Figure 4.12.

### Mappers and Transformers

So far we have only looked at the data mapping between the “employee” form and its children. We did not look yet at the mapping between the “birthDate” form and its children “year”, “month” and “day”. In fact, this case is a bit different, because the data of that form, a `DateTime` object, neither offers the accessors `getYear`, `getMonth` and `getDay` nor the modifiers `setYear`, `setMonth` or `setDay`. Even though it offers generic accessors and modifiers such as `setDate`, I prefer to treat the `DateTime` object as immutable and create a new instance for every new date value.
An elegant solution to this problem is the combination of data mappers with the previously described data transformers. At first, a data transformer converts the DateTime instance to an array with the three keys “year”, “month” and “day”. This array can then be mapped to the form’s children and back. After updating the array, the new array can be reverse-transformed into a new DateTime instance, which is stored in the form. Figures 4.13 and 4.14 illustrate this approach.

The conclusion of the last example is that the data mapped to the children of a form must always be in the view representation $V$, which has already been processed by a transformer.

---

An algorithm and architecture for server-side processing of model-bound compositional web forms
4.6 Events

Events provide a flexible mechanism for executing custom code at predefined points during the program flow of an application. These predefined points are identified by names $e$. Developers may write any number of event listeners $l_0, \ldots, l_k$ and connect them to $e$. Whenever the point identified by $e$ is reached in the program flow, an event is dispatched, leading to $l_0, \ldots, l_k$ being notified. Typically, listeners are notified in the same order in which they have been connected, but in some event listener implementations it is also possible to prioritize listeners.

```
+dispatch($eventName, Event $event = null)
+addListener($eventName, $listener, $priority = 0)
+removeListener($eventName, $listener)
```

Figure 4.15: The interface `EventDispatcher` in Symfony2.

Figure 4.15 demonstrates an interface `EventDispatcher` as implemented in Symfony2. It supports the following methods:

- `dispatch`: Notifies all listeners connected for the event `$eventName`. Optionally an `Event` object can be passed.
- `addListener`: Adds a listener to `$eventName`. Listeners can be any kind of PHP callables, for example a closure or a method of an object. Optionally, the listener can be prioritized: Listeners with higher priorities are called before listeners with lower priorities.
- `removeListener`: Removes a listener from `$eventName`.

Events in the binding and prepopulation process of a form allow to solve a number of problems mentioned in chapter 3 in an elegant fashion. The following sections explore these solutions in detail.

Validation

When a form is submitted to the server, the content of its fields typically needs to be validated against a number of constraints. For example, for some fields the user might be required to enter a value while for others not. Or he might be required to submit a value in a specific format or within a specific domain, like a date in the past. If a user submits a form violating these constraints, the form needs to be displayed again. Appropriate error messages should be shown in order to indicate how to correct the form before submitting it again.

Consequently, validation must be launched during the binding process. The best moment for doing so is at the end of the `bind` method, when the reverse transformation process is complete. Section 4.3 explained that the transformation process may fail, resulting in a form whose data is not synchronized. When this happens, (I) the form must be marked with an error stating that the form’s value is invalid. Additionally, (II) any further validation errors must be ignored.

The reasoning for rule (I) should be obvious. If the user submitted a value that cannot be reverse-transformed, that value needs to be corrected.

The reasoning for rule (II) is that the validation might depend on the form’s data in any of the representations $V$, $N$ or $M$. However, if a form is not synchronized, this means that the semantics of the values in the three representations do not match, potentially leading to false negatives.
In order to decouple actual form validation from the form’s processing logic, validation can be outsourced into an event listener. This listener can be connected to the event \texttt{POST\_BIND} that is dispatched at the end of the \texttt{bind} method. In order to attach errors to a form, the following methods need to be added to \texttt{Form}:

- \texttt{addError}: Attaches an error to the form.
- \texttt{getErrors}: Returns the errors attached to the form.
- \texttt{hasErrors}: Returns whether the form has any errors.
- \texttt{isValid}: Returns \texttt{false} if the form or any of its descendants in the form tree has errors, \texttt{true} otherwise.

The benefit of decoupled validation by means of an event listener is that the listener can easily be replaced by a custom implementation. This allows developers to integrate whatever validation library fulfills best the needs of their application.

\section*{Data Cleaning}

Data cleaning refers to the solution of the \textit{Dirty Data} problem. Dirty data, within the context of this work, is data that is semantically correct but syntactically malformed. For example, the user might not be allowed to type leading or trailing spaces in text fields. Or the user might be expected to always start a URL with the characters “http://”.

One way for solving this problem is to simply disallow such inputs and to show an error message, requesting the user to correct the form. Such restrictions, however, make it hard for less experienced users to submit a valid form. Also, error tolerance has been identified as one of the key issues to dialog usability [15]. Consequently, forms should try to automatically correct the syntax of the submitted values if they are able to do so. The removal of leading or trailing whitespace, for example, is a trivial task in most programming languages. Also, the correction of a URL to start with “http://”, if it does not already, is trivial to implement.

Within \texttt{Form}, automatic input correction can be achieved by an event that is dispatched at the beginning of the \texttt{bind} method. The event needs to provide the data submitted by the client and allow to change the data before continuing form processing. This event will be called \texttt{PRE\_BIND} here.

Likewise, a second event \texttt{BIND} can be introduced that is dispatched once the reverse transformation to the normalized representation \(N\) is complete. This event allows data cleaning in those cases where the view representation \(V\) is dynamic.

\section*{Dynamic Forms}

As described in section 3.4, web servers increasingly need to adapt form processing to dynamic changes in the Document Object Model (DOM) made by JavaScript code. In the description of the \textit{Dynamic Forms} problem I have already given an example of a profile having an arbitrary number of email addresses. Commonly, such interfaces feature JavaScript-powered buttons that add additional or remove existing form fields for entering such addresses. The form model on the server needs to adapt itself when a dynamically changed form is submitted. For example, when a new email field is added to the DOM, a new corresponding \texttt{Form} instance needs to be inserted into the server-side form tree before launching \texttt{bind} in order to ensure correct form processing.
Events provide an elegant solution to this problem. The event `PRE_BIND` at the beginning of the `bind` method can be reused to introspect the data and adjust the form tree before letting the normal control flow proceed.

A list of email addresses usually is also represented by a list type on the server, for example as array. When constructing a form for the first time, an email field needs to be inserted in the form tree for every entry in the array. This use case can be solved like the previous one. An event `PRE_SET_DATA` is dispatched at the beginning of the `setData` method that dynamically adapts the form before mapping the passed data to the form’s children.

### 4.7 Summary

This section gives an overview of the concepts presented in this chapter. At the beginning, the compositional interface `Form` was introduced which represents elements of a form in a tree-like structure. Each of these elements, which I simply call forms, is connected with two data transformer chains, an event dispatcher and optionally a data mapper. Figure 4.16 illustrates these associations.

![Figure 4.16: Form and its associations.](image)

The first data transformer chain contains the normalization transformers which convert the value of a form from the representation `M` used in the domain model to the normalized representation `N` used within the form object. The second chain contains the view transformers which convert the normalized value to the representation `V` used in the view. The data mapper, which is only required if the form has children, maps the data of a form to these children. For example, an array-based data mapper expects a form’s data to be an array and passes the individual entries of the array to the children of the form. The event dispatcher, at last, manages all listeners connected to the form and notifies them at specific points during the binding and prepopulation processes.

### Life Cycle

Figure 4.17 depicts the life cycle of a `Form` object. When the form object is first created, it enters the state `New`. Its data is set to `null` at this point. Subsequently the developer may call either `setData`, to prefill the form with a default value, or `bind`, to update the form with data submitted by the browser. If `setData` is called, the form enters the state `Prepopulated`. At the beginning of this state, the passed data is transformed to the normalized representation `N` and to the view representation `V`. If, on the other hand, `bind` is called, the form enters the state `Bound`. When this state is entered, the bound data is reverse-transformed to the representations `N` and `M`. If any of these transformations fails due to illegal submitted values, the form is considered not synchronized and immediately enters the state `Invalid`. If reverse transformation...
succeeds and the form is the root of the form tree, the whole form tree is validated by means of a
POST_BIND event listener. Errors discovered during the validation are added to the appropriate
forms in the tree, which then also enter the state Invalid. More than one error can be added to
each node, but only if the node’s data is synchronized (see section 4.6).

Prepopulation

Forms can be prepopulated with a default value by calling the method setData. Figure 4.18
summarizes the control flow of this method.

Figure 4.17: The life cycle of the interface Form.

Figure 4.18: The control flow of setData.
The method `setData` expects a single argument, which is the default data in the model representation $M$. At first, the event `PRE_SET_DATA` is dispatched. Listeners for this event receive the passed default data and can modify it, if desired. The resulting data is then transformed to the normalized representation $N$ and then to the view representation $V$. Then, the data is stored in the form in all three representations. The form is additionally marked as synchronized. Next, the form’s data is mapped to its children, if it contains any. The data is passed to the mapper in the view representation $V$ for reasons described in section 4.5. At last, the event `POST_SET_DATA` is dispatched, which has not been mentioned previously, but is included here for sake of completeness. The data is passed to listeners of this event in representation $M$.

**Binding**

A form tree can be updated with data submitted by a client by passing this data to the `bind` method. Figure 4.19 illustrates the control flow of this method.

![Diagram of the control flow of the `bind` method](image)

Figure 4.19: The control flow of `bind`.

Similar to `setData`, the event `PRE_BIND` is dispatched at the beginning of `bind`. It allows to read and modify the data submitted by the client. The resulting data is then passed to the child forms by calling their `bind` methods. After binding the children, their reverse-transformed values are mapped back into the data of the form. As explained in section 4.5, the data must be passed to the mapper in the view representation $V$. Then, the mapped data is reverse-transformed to the normalized representation $N$. If an exception is thrown in this process, transformation cannot continue. Instead, the updated value is stored only in the view representation. The form is additionally marked as not synchronized. If, on the other hand, transformation to $N$ succeeded, the normalized data is further passed to listeners of the `BIND` event, which have the opportunity to change it. Finally, the data is reverse-transformed to the model representation $M$. If everything goes well, the form is marked as synchronized. In any case, the `POST_BIND` event is dispatched and allows for example validation to take place.
Chapter 5. Conclusion

The previous chapters presented a catalog of web form-related problems and an architecture that solves these problems. This chapter provides a summary of these chapters and gives an example of using this architecture in practice. Section 5.1 summarizes the results of the thesis and relates them to its original goals. Section 5.2 discusses whether and how the problems in chapter 3 have been solved. Section 5.3 gives an example of how this architecture is implemented and used in the web framework Symfony2. Section 5.4 explains what challenges remain to be solved in the future. Section 5.5 finally concludes the lessons learned throughout this thesis.

5.1 Results

This thesis presents an architecture for processing web forms following the four major goals described in section 1.2:

- Reusability is enabled by separating the different responsibilities of web forms into forms, data transformers, data mappers and event listeners that can be assembled and reused with a fitting abstraction mechanism. The description of such an abstraction mechanism is subject to future work.
- Such forms can be composed to model forms of a higher complexity.
- Crosscutting is reduced by centralizing structural information in the form tree and reusing it elsewhere instead of duplicating it.
- Model binding is partially solved by reading and writing the model autonomously using data mappers. The description of reusing the structural information of the domain model needs to be explained in future papers.

As a foundation for the design decisions taken in this thesis, chapter 3 analyzes common problems faced by form processing libraries. Chapter 4 then presents a three-step approach to solving these problems. The first of these three steps is described in detail in the remainder of that chapter. The last two steps remain to be described in future articles that contribute the missing pieces to achieving the above goals.

5.2 Discussion

Chapter 4 suggests a step-by-step approach for solving the problems listed in chapter 3 and continues to present a design for the first step, “Generic Form Model and Algorithm Design”. This section discusses how the proposed design actually solves the problems associated with this step.

Name Mismatch

Each field in the form is represented by an object that generates a unique HTML name for that field (see section 4.4). This name is both printed in the “name” attribute of the field’s HTML output and used for finding the matching CGI parameter and binding it back to the object. Thus the Name Mismatch problem is solved.
**Impedance Mismatch**

The usage of data transformers solves the *Impedance Mismatch* problem. When a field’s object is populated with a value from the domain model, this value is transformed to the data type used in the HTML output. When a value is submitted and bound to the object, the data transformer transforms the submitted value back before it is written into the domain model (see section 4.3).

**Dirty Data**

The *Dirty Data* problem is solved by attaching an event listener to the `PRE_BIND` event of the form that removes undesired input from the submitted values (see section 4.6).

**Invalid Data**

Data is validated in an event listener attached to the `POST_BIND` event of the form. If the form’s data does not match the predefined constraints, errors can be attached to the form which are shown when the form is displayed again (see section 4.6).

**Name Clashes**

Each object in the form tree has a name that is unique within its siblings. The HTML name of a form is the concatenation of the its name and the names of all of its ancestors (see section 4.4). Consequently, even if two forms in the form tree have the same name, no two forms can have the same HTML name. The *Name Clashes* problem is solved.

**Model Update and Prepopulation**

The data mapper solves the *Model Update* and *Prepopulation* problems. Instances of the domain model can be passed as default values to the form. The data mapper then extracts the information that should be displayed in the form as default values. Once the form is submitted, the data mapper writes the submitted values back into the domain model instances (see section 4.5).

**Compositionality**

Both fields and forms are composed in a tree of instances implementing the same interface. Each instance completely encapsulates its own responsibilities. Forms and fields can thus be freely composed to higher-level types (see section 4.2).

**Dynamic Forms**

The introduction of events in the prepopulation and binding process allows to create highly dynamic forms. Listeners to the `PRE_BIND` event can introspect the submitted values and adjust the form tree accordingly. The same can be achieved in the prepopulation process by listening to the `PRE_SET_DATA` event (see section 4.6).

### 5.3 Usage Example

The three-step approach described in section 4.1 is implemented in the PHP framework Symfony2. In Symfony2, forms can be abstracted in so-called *form types*. Listing 5.1 shows the form type code for the form in section 2.1.

```php
namespace ExampleBundle\Form;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\ FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;
```
class EmployeeType extends AbstractType
{
    public function getName()
    {
        return "employee";
    }

    public function getParent()
    {
        return "form";
    }

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            "data_class" => "Employee",
        ));
    }

    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add("name", "text")
            ->add("salary", "money", array(
                "divisor" => 100,
            ))
            ->add("birthDate", "birthdate")
    }
}

Listing 5.1: The form type EmployeeType.

For convenience reasons, form types usually extend AbstractType which implements the methods of FormTypeInterface as empty stubs. Each type has a unique name, which is returned by the method getName. Each type can also have a supertype, whose name is returned by getParent. The base type for all types without an explicit supertype is “form”. This type is implemented in Symfony2 and contains all logic shared among all types.

Each type may furthermore define options that can be set by the developer using the type. This is done in the method setDefaultOptions. Options can be used to customize a type, for example its rendering, its functionality or its label. By default, each type inherits the options of its supertypes. In this example, EmployeeType inherits all options of the “form” type. The option “data_class” is one of these options and configures the class of the object manipulated by the form.

The method buildForm, at last, contains the actual configuration of the form. This configuration is done using a form builder, which produces a tree of Form instances, once ready.

Form types need to be registered in the dependency injection container. Listing 5.2 shows the necessary XML configuration for registering the EmployeeType. The type is registered as simple service and must be tagged with the tag "form.type". The alias of the tag must match the type’s name.

<?xml version="1.0" ?>
<container xmlns="http://symfony.com/schema/dic/services"
The registered form type can finally be used in the controller. Listing 5.3 shows a sample Symfony2 controller using the form. Each controller in Symfony2 may extend the base class Controller for convenience reasons and contains one or more actions. In this case, the controller contains only one action, newAction, for creating new employees. This method creates a form of type “employee” on line 10. If the form has not been submitted yet, the action continues to render the template “new.html.twig” on line 30, passing a tree of FormView instances in the variable “form”, which is obtained by calling createView on the form.

Once the form is submitted, the if-condition on line 12 is satisfied. The form is then bound with the submitted data on line 13. If no validation error occurred, the created Employee instance is extracted from the form on line 16. The instance is persisted using the Object-Relational Mapping (ORM) Doctrine on lines 18-20 before redirecting to a page that indicates the successful submission of the form on line 24.

Listing 5.2: The XML configuration for the dependency injection container.

Listing 5.3: A sample Symfony2 controller using the form.
Listing 5.3: The Symfony2 controller processing the employee form.

Listing 5.4 shows the code of the template “new.html.twig”. The template uses the templating library Twig\(^2\), a PHP library inspired by Python’s templating engine Jinja\(^3\).

The template renders the whole form using Symfony’s default theme, which defines the layout of form labels, errors and fields. The rendering can also easily be customized. More information about that can be found in the online documentation\(^4\).

Listing 5.4: The Twig template containing the HTML.

5.4 Future Work

Section 4.1 presented a three-step approach to building a form library, of which the first step has been elaborated in detail in this thesis. While all three steps have been implemented in the Symfony2 framework, a detailed theoretical coverage is missing for step two and three. This coverage is subject to future work.

Furthermore, the application of the reference implementation in Symfony2 revealed some problems with the current design. The first problem is the missing support for creating fields dependent on the default or submitted values of other fields. A classical example is a form with three drop down fields for selecting a country, a province and a city. The province drop down field should display only provinces of the selected country, so it needs to be created after binding the country field. The same applies for the city field with regard to the province field. While this currently can be implemented to some extent by using events, such an implementation is complicated and flawed. Consequently, explicit support for this feature should be added to future versions of the architecture.

Last, the current design did not involve the automatic generation of JavaScript code in order to enhance the usability and speed of forms. An analysis is required that discusses the possibilities of such a feature. Future versions of the Symfony2 forms should support JavaScript in one way or another to support truly rapid development of state-of-the-art web forms.

\(^2\) http://twig.sensiolabs.org
\(^3\) http://jinja.pocoo.org
5.5 Conclusion

In summary it can be said that the creation of an architecture for processing web forms requires a considerable amount of effort, analysis and design. While a couple of form processing libraries exist in the academia and many more in the industry, many fail to properly address all the different problems faced when creating modern enterprise web applications. The biggest of these problems is structural crosscutting, which is inherent to the nature of web forms and Model-View-Controller designs. The second big problem that many of these libraries fail to solve is to properly enable reusability of form components. This thesis approaches and solves these problems. While the presented solution certainly has flaws that remain to be discovered and solved, I am confident that it is an important step towards simple, reusable, compositional and model-bound web forms.
List of Figures

2.1 A simple form for entering the personal data of an employee. . . . . . . . . . . . . 4
2.2 The form of Figure 2.1 after an invalid submission. . . . . . . . . . . . . . . . . . 5

4.1 An intuitive model for the example in section 2.1. . . . . . . . . . . . . . . . . . . 19
4.2 An improved version of the model in Figure 4.1. . . . . . . . . . . . . . . . . . . 20
4.3 The compositional interface Form. . . . . . . . . . . . . . . . . . . . . . . . . . 20
4.4 The interface for transforming data between $M$ and $V$. . . . . . . . . . . . . . . 21
4.5 Transformation from $M$ to $V$ when invoking $\text{setData}$. . . . . . . . . . . . . . . 22
4.6 Reverse transformation from $V$ to $M$ when invoking $\text{bind}$. . . . . . . . . . . . . . 22
4.7 Data distribution during binding. . . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.8 Simplified data distribution by passing only relevant data. . . . . . . . . . . . . . . 27
4.9 Update of the $\text{Employee}$ object after binding. . . . . . . . . . . . . . . . . . . 27
4.10 Data distribution after setting the default data. . . . . . . . . . . . . . . . . . . . 28
4.11 The interface for mapping a form’s data onto its children and back. . . . . . . . . . 28
4.12 Dynamic mapping between form models and domain models in Symfony2. . . . . 29
4.13 Distributing a $\text{DateTime}$ instance by first transforming it to an array. . . . . . . 29
4.14 Recollecting the values of the transformed $\text{DateTime}$ instance. . . . . . . . . . . 29
4.15 The interface $\text{EventDispatcher}$ in Symfony2. . . . . . . . . . . . . . . . . . . 30
4.16 Form and its associations. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.17 The life cycle of the interface Form. . . . . . . . . . . . . . . . . . . . . . . . . 33
4.18 The control flow of $\text{setData}$. . . . . . . . . . . . . . . . . . . . . . . . . . 33
4.19 The control flow of $\text{bind}$. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
List of Tables

4.1 The domains of the salary field. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
4.2 The domains of the customized money field. . . . . . . . . . . . . . . . . . . . . . 23
4.3 A money field with configurable domains. . . . . . . . . . . . . . . . . . . . . . . 24
4.4 The money field with the additional normalized domain. . . . . . . . . . . . . . . 24
## List of Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A sample HTTP POST request of <code>newEmployee.php</code>.</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>A dynamic HTML template created with PHP.</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>The domain model class <code>Employee</code>.</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>The template <code>newEmployee.php</code>.</td>
<td>11</td>
</tr>
<tr>
<td>3.2</td>
<td>The controller for processing the form in listing 3.1.</td>
<td>12</td>
</tr>
<tr>
<td>3.3</td>
<td>The <code>Employee</code> class with constrained properties.</td>
<td>14</td>
</tr>
<tr>
<td>4.1</td>
<td>Example CGI parameters sent by the form in section 2.1.</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>The CGI parameters generated by the new naming scheme.</td>
<td>26</td>
</tr>
<tr>
<td>4.3</td>
<td>PHP’s interpretation of the parameters in Listing 4.2.</td>
<td>26</td>
</tr>
<tr>
<td>5.1</td>
<td>The form type <code>EmployeeType</code>.</td>
<td>36</td>
</tr>
<tr>
<td>5.2</td>
<td>The XML configuration for the dependency injection container.</td>
<td>37</td>
</tr>
<tr>
<td>5.3</td>
<td>The Symfony2 controller processing the employee form.</td>
<td>38</td>
</tr>
<tr>
<td>5.4</td>
<td>The Twig template containing the HTML.</td>
<td>39</td>
</tr>
</tbody>
</table>
Bibliography


Appendix


Appendix


