2 Ключевые понятия

Паттерны проектирования — это проверенные решения типичных задач разработки, помогающие улучшить организацию и структуру приложений. Используя паттерны, мы применяем коллективный опыт квалифицированных разработчиков в решении подобных задач.

Разработчики настольных и серверных приложений уже давно пользуются многочисленными паттернами проектирования, однако в разработке клиентских приложений подобные паттерны стали применяться лишь несколько лет назад.

В этой главе мы изучим развитие паттерна проектирования «Модель-Представление-Контроллер» (Model-View-Controller, MVC) и узнаем, как библиотека Backbone.js использует этот шаблон в разработке клиентского приложения.

Паттерн MVC

MVC представляет собой паттерн проектирования архитектуры, который улучшает структуру приложения путем разделения его задач. Он позволяет изолировать бизнес-данные (модели) от пользовательских интерфейсов (представлений) с помощью третьего компонента (контроллеров), который управляет логикой и вводом пользовательских данных, а также координирует модели и представления. Исходный паттерн был создан Трюгве Реенскаугом (Trygve Reenskaug) в 1979 году во время его работы над языком Smalltalk-80 и назывался «Модель-Представление-Контроллер-Редактор» (Model-View-Controller-Editor). Паттерн MVC подробно описан в книге «Банды Четырех» (Gang of Four) под названием «Приемы объектно-ориентированного проектирования. Паттерны проектирования» (Design Patterns: Elements of Reusable Object- Oriented Software), вышедшей в 1994 году и способствовавшей популяризации его использования.

MVC в Smalltalk-80

Важно понимать задачи, для решения которых разрабатывался исходный паттерн MVC, поскольку на сегодняшний день круг этих задач существенно изменился. В 1970-е годы графические пользовательские интерфейсы еще не получили широкого распространения. Для отделения объектов, моделировавших предметы реального мира (например, фотографии и людей), от объектов представления, которые выводились на экран пользователя, стал использоваться подход под названием *раздельное отображение*.

Реализация паттерна MVC в языке Smalltalk-80 развила эту концепцию и ее целью стало отделение логики приложения от пользовательского интерфейса. Идея заключалась в том, что разделение приложения на эти части позволит повторно использовать модели в других интерфейсах приложения. В MVC-архитектуре языка Smalltalk-80 есть несколько интересных аспектов, о которых стоит упомянуть:

- Доменный элемент назывался моделью и не имел никакой связи с пользовательским интерфейсом (представлениями и контроллерами).
- За отображение отвечали представление и контроллер, но не в единственном экземпляре: для каждого элемента, отображаемого на экране, требовалась пара «представление–контроллер», и между ними фактически не существовало разделения.
- Роль контроллера в этой паре заключалась в обработке пользовательского ввода, например событий, соответствующих нажатиям клавиш и щелчкам кнопкой мыши, и выполнению каких-либо осязаемых действий над ними.
- Паттерн наблюдателя использовался для обновления представления при изменении модели.

Иногда разработчики удивляются, узнав, что паттерн наблюдателя (который на сегодняшний день обычно реализуется в виде модели публикации/подписки) был частью архитектуры MVC несколько десятилетий назад. В паттерне MVC языка Smalltalk-80 за моделью наблюдают и представление, и контроллер: представление реагирует на каждое изменение модели. Простой пример этого подхода — приложение для фондовой биржи: чтобы оно могло отображать информацию в реальном времени, любое изменение данных в его модели должно приводить к немедленному обновлению представления.

Мартин Фаулер (Martin Fowler) написал отличную работу об истоках MVC; если вас интересует дополнительная информация о развитии MVC в языке Smalltalk-80, я рекомендую прочитать ее.

MVC во Всемирной паутине

Всемирная паутина в значительной степени опирается на протокол HTTP, у которого нет *внутреннего состояния*. Это означает, что не существует постоянно открытого соединения между браузером и сервером: каждый запрос приводит к созданию нового канала связи между ними. Как только сторона, инициировавшая запрос (например, браузер), получает ответ, соединение закрывается. Такой контекст отличается от контекста операционных систем, в котором разрабатывались многие изначальные идеи MVC. Реализация MVC должна соответствовать веб-контексту.

Примером фреймворка для серверных веб-приложений, в котором сделана попытка применения MVC-подхода в контексте Всемирной паутины, является *Ruby on Rails* (рис. 2.1).

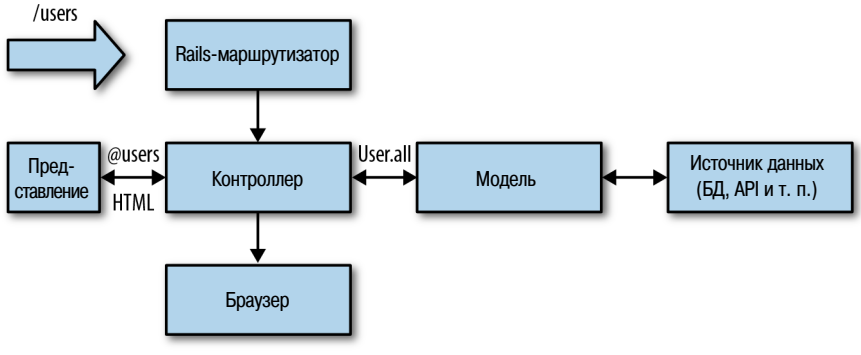

Рис. 2.1. Фреймворк Ruby on Rails

Ядро этого фреймворка составляет архитектура на основе трех знакомых нам MVC-компонентов: модели, представления и контроллера. В Rails они имеют следующий смысл:

- Модели отражают данные приложения и используются для управления правилами взаимодействия со специальной таблицей базы данных. Как правило, одна таблица соответствует одной модели, а модели содержат основную часть бизнес-логики приложения.
- Представления отражают пользовательский интерфейс, часто в виде HTML-кода, который передается в браузер. Они представляют данные приложения стороне, которая запрашивает их.
- Контроллеры являются связующим звеном между моделями и представлениями. Они ответственны за обработку запросов от браузера, обращение

к моделям для считывания данных, а также за передачу этих данных в представления для последующего отображения в браузере.

Несмотря на то что Rails имеет схожее с MVC распределение задач приложения, на самом деле, в Rails используется другой паттерн под названием *Model2*. Это объясняется тем, что в Rails представления не получают уведомлений от моделей, а контроллеры просто передают данные модели напрямую в представление.

Вместе с тем, генерация HTML-страницы в качестве ответа и отделение бизнес-логики от интерфейса дает множество преимуществ даже при приеме URL-запроса серверной стороной приложения. Преимущества, которые предоставляет четкое отделение бизнес-логики от записей базы данных в серверных фреймворках, эквивалентны преимуществам четкого разделения между пользовательским интерфейсом и моделями данных в JavaScript (об этом мы далее поговорим подробнее).

Другие серверные паттерны MVC, например PHP-фреймворк *Zend*, также реализуют паттерн проектирования контроллера запросов. Этот паттерн располагает MVC-стек за единственной точкой входа, благодаря которой все HTTP-запросы (такие, как *http://www.<имя>.com*, *http://www.<имя>.com/<страница>/* и другие) направляются конфигурацией сервера одному и тому же обработчику независимо от URI.

Когда контроллер запросов получает HTTP-запрос, он анализирует его и принимает решение о том, какой класс (контроллер) и метод (действие) вызвать. Выбранное действие контроллера получает управление и взаимодействует с соответствующей моделью для выполнения запроса. Контроллер получает данные от модели, загружает соответствующее представление, вставляет в него данные модели и возвращает ответ браузеру.

Допустим, что у нас есть блог по адресу www.example.com, мы хотим отредактировать на нем статью с идентификатором 43 (id=43) и посылаем запрос http://www.example.com/article/edit/43.

На стороне сервера контроллер запросов проанализирует URL и вызовет контроллер статьи (который соответствует части /article/ URI) и его редактирующее действие (которое соответствует части /edit/ URI). Внутри действия будет обращение к модели статей (назовем ее так) и ее методу Articles::getEntry(43) (*43* соответствует части */43* URI). В ответ будут возвращены данные статьи блога для редактирования. Затем контроллер статьи загрузит представление article/edit, содержащее логику для вставки данных статьи в форму и позволяющее отредактировать содержание, заголовок статьи и другие (мета)данные. В конце результирующий HTML-ответ будет возвращен в браузер.

Как вы можете догадаться, аналогичная процедура необходима и для обработки запросов POST, когда мы щелкаем по кнопке сохранения в форме. URI для POST-действия будет иметь вид /article/save/43. Запрос будет передан тому же контроллеру, но на этот раз будет вызвано действие сохранения (из-за фрагмента /save/ URI), модель статей сохранит отредактированную статью в базе данных с помощью вызова Articles::saveEntry(43) и браузер будет перенаправлен по URI /article/edit/43 для дальнейшего редактирования статьи.

Наконец, при введении запроса http://www.example.com/ контроллер запросов вызовет контроллер и действие по умолчанию (например, индексный контроллер и индексное действие). Внутри этого действия будет обращение к модели статей и ее методу Articles::getLastEntries(10), который вернет последние 10 публикаций блога. Контроллер загрузит представление blog/index с базовой логикой отображения публикаций.

На рис. 2.2 показан типичный жизненный цикл запросов/ответов HTTP для серверной части MVC.

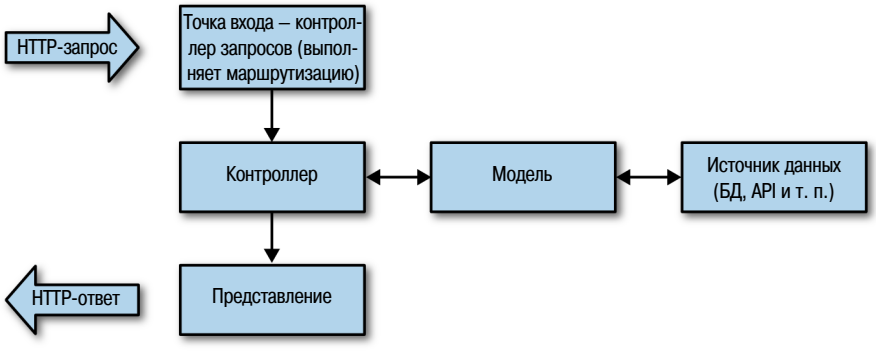

Рис. 2.2. Жизненный цикл запросов/ответов HTTP для серверной части MVC

Сервер получает HTTP-запрос и направляет его через единственную точку входа, на которой контроллер запросов анализирует запрос и вызывает действие соответствующего контроллера. Этот процесс называется *маршрутизацией*. Модель действия получает запрос на возврат и/или сохранение переданных данных. Эта модель взаимодействует с источником данных (например, с базой данных или API). Когда модель заканчивает работу, она возвращает данные контроллеру, который затем загружает соответствующее представление. Это представление выполняет логику отображения (перебирает статьи и печатает их заголовки, содержимое и прочее) с использованием предоставленных данных. В конце HTTP-ответ возвращается браузеру.

MVC на клиентской стороне и одностраничные приложения

Ряд исследований подтвердил, что сокращение задержек при взаимодействии пользователей с сайтами и приложениями положительно влияет на их популярность, а также на степень вовлеченности их пользователей. Это противоречит традиционному подходу к разработке веб-приложений, который сконцентрирован на серверной части и требует полной перезагрузки страницы при переходе с одной страницы на другую. Даже при наличии мощного кэша браузер должен выполнить разбор CSS, JavaScript и HTML, а также отобразить интерфейс на экране.

Такой подход не только приводит к повторной передаче пользователю большого объема данных, но и отрицательно влияет на задержки и общую быстроту взаимодействия интерфейса с пользователем. В последние годы для уменьшения ощущаемых пользователем задержек разработчики зачастую создают одностраничные приложения, которые сначала загружают единственную страницу, а затем обрабатывают навигационные действия и запросы пользователей, не перезагружая страницу целиком.

Когда пользователь переходит к новому представлению, приложение обращается за дополнительным содержимым для этого представления с помощью запроса *XHR* (XMLHttpRequest), как правило, к серверному *REST API* или конечной точке. *AJAX* (Asynchronous JavaScript and XML, асинхронный JavaScript и XML) обеспечивает асинхронное взаимодействие с сервером, при котором передача и обработка данных происходит в фоновом режиме без вмешательства в работу пользователя с другими частями страницы. Это делает интерфейс более быстрым и удобным.

Одностраничные приложения могут использовать такие возможности браузера, как API журнала для обновления поля адреса при перемещении из одного представления в другое. Эти URL также позволяют пользователям создавать закладки на определенных состояниях приложения и обмениваться ими без необходимости загружать совершенно новые страницы.

Типичное одностраничное приложение содержит небольшие логические элементы с собственными пользовательскими интерфейсами, бизнес-логикой и данными. Хорошим примером является корзина, в которую можно добавлять товары в веб-приложении интернет-магазина. Эту корзину можно отображать в правом верхнем углу страницы (рис. 2.3).

Корзина и ее данные представлены в виде HTML. Данные и связанное с ними представление изменяются с течением времени. Когда-то мы использовали jQuery (или похожую DOM-библиотеку) и множество прямых и обратных AJAX-вызовов для синхронизации данных и представления. В результате

получался плохо структурированный код, который было трудно поддерживать. Ошибки были частыми, а зачастую и неизбежными.

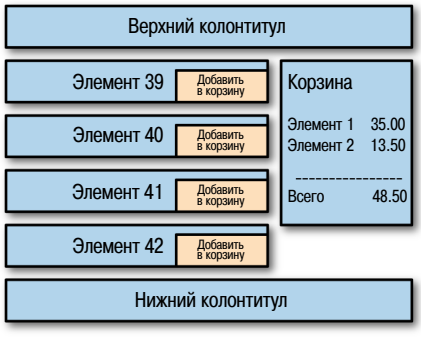

Рис. 2.3. Товарная корзина, формирующая регион одностраничного приложения

Потребность в динамичных и сложных веб-приложениях с быстро реагирующими интерфейсами на основе AJAX потребовала дублирования значительной части описанной логики на стороне клиента, что резко увеличило размер и сложность кода клиентской части приложения. В конечном счете, возникла необходимость реализовать на клиентской стороне MVC или другую схожую архитектуру, чтобы улучшить организацию кода, упростить его поддержку и продлить жизненный цикл приложения.

JavaScript-разработчики воспользовались преимуществами традиционного паттерна MVC и с помощью проб и ошибок постепенно создали несколько MVC-подобных JavaScript-фреймворков, таких как Backbone.js.

MVC на клиентской стороне: стиль Backbone

Давайте рассмотрим, как библиотека Backbone.js обеспечивает преимущества паттерна MVC при разработке клиентской части на примере приложения для управления задачами. Мы будем расширять этот пример в следующих главах по мере изучения возможностей Backbone, однако сейчас сконцентрируем внимание на том, как ключевые компоненты этого приложения соотносятся с MVC.

В данном примере нам потребуется элемент div, к которому мы привяжем список задач, а также HTML-шаблон, содержащий место для этого списка, и флажок завершения, который можно устанавливать для отдельных задач. Все перечисленные компоненты создаются при помощи следующего HTML-кода:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title></title>
    <meta name="description" content="">
</head>
<body>
    <div id="todo">
    </div>
    <script type="text/template" id="item-template">
        <div>
            <input id="todo_complete" type="checkbox" <%= completed ?
                'checked="checked"' : '' %>>
            <%- title %>
        </div>
    </script>
    <script src="jquery.js"></script>
    <script src="underscore.js"></script>
    <script src="backbone.js"></script>
    <script src="demo.js"></script>
</body>
</html>
```

В нашем приложении (в файле demo.js) данные каждой задачи хранятся в экземплярах моделей Backbone:

```
// определение модели Todo
var Todo = Backbone.Model.extend({
    // значения атрибутов todo по умолчанию
    defaults: {
        title: '',
        completed: false
    }
});
// Создание экземпляра модели задачи с помощью ее названия,
// атрибут completed имеет значение false по умолчанию
var myTodo = new Todo({
    title: 'Check attributes property of the logged models in the console.'
});
```

Наша модель задачи расширяет модель Backbone.Model и просто определяет значения по умолчанию для двух атрибутов данных. В следующих главах вы увидите, что возможности моделей Backbone гораздо шире, однако этот простой пример иллюстрирует, что модель в первую очередь является контейнером для данных.

Каждая задача будет отображаться на странице с помощью представления TodoView:

```
var TodoView = Backbone.View.extend({
    tagName: 'li',
    // Кэширование функции шаблона для отдельной задачи.
    todoTpl: _.template( $('#item-template').html() ),
    events: {
        'dblclick label': 'edit',
        'keypress .edit': 'updateOnEnter',
        'blur .edit': 'close'
    },
    // Вызывается при первом создании представления
    initialize: function () {
        this.$el = $('#todo');
        // Позже мы рассмотрим вызов
        // this.listenTo(someCollection, 'all', this.render);
        // но вы можете запустить этот пример прямо сейчас,
        // вызвав метод TodoView.render();
    },
    // Повторное отображение заголовков задач.
    render: function() {
        this.$el.html( this.todoTpl( this.model.toJSON() ) );
        // Здесь $el - это ссылка на элемент jQuery,
        // связанный с представлением, todoTpl - ссылка
        // на Underscore-шаблон,а toJSON() возвращает объект,
        // содержащий атрибуты модели.
        // В совокупности этот оператор заменяет HTML-код
        // DOM-элемента на результат создания
        // экземпляра шаблона с атрибутами модели.
        this.input = this.$('.edit');
        return this;
    },
    edit: function() {
        // выполняется при двойном щелчке по ярлыку задачи
    },
    close: function() {
        // выполняется, когда задача теряет фокус
    },
    updateOnEnter: function( e ) {
        // выполняется при каждом нажатии клавиши в режиме редактирования задачи,
        // но мы будем ждать нажатия enter, чтобы перейти в действие
    }
});
// создание представления задачи
var todoView = new TodoView({model: myTodo});
```

Мы определяем представление TodoView, расширяя класс Backbone.View, и создаем его экземпляр представления с помощью соответствующей модели. В нашем примере метод render() использует шаблон для формирования HTML-кода задачи внутри элемента li. Каждый вызов функции render() замещает содержимое элемента li данными текущей модели. Таким образом, экземпляр представления отображает содержимое DOM-элемента, используя атрибуты связанной с ним

модели. Позднее мы увидим, как представление может связать свой метод render() с событиями изменения модели, которые приводят к повторному отображению представления при изменении модели.

Итак, мы увидели, что с точки зрения паттерна MVC класс Backbone.Model реализует модель, а Backbone.View — представление. Однако, как мы отмечали ранее, библиотека Backbone отличается от традиционного MVC в том, что касается контроллеров, — класса Backbone.Controller не существует!

Вместо него функции контроллера выполняет представление. Вспомните, что контроллеры реагируют на запросы и выполняют соответствующие действия, которые могут привести к изменению модели и обновлению представления. В одностраничном приложении вместо запросов в традиционном смысле этого термина используются события. События могут включать в себя обычные DOM-события (такие, как щелчки) и внутренние события приложения (такие, как изменения модели).

В нашем представлении TodoView атрибут events играет роль конфигурации контроллера, которая определяет, как события, происходящие внутри DOM-элемента представления, должны передаваться методам обработки событий, определенным в представлении.

Хотя в этом примере события помогают нам связать библиотеку Backbone с паттерном MVC, мы увидим, что они будут играть бóльшую роль в наших одностраничных приложениях. Класс Backbone.Event — это фундаментальный компонент Backbone, который используется как в классе Backbone.Model, так и в классе Backbone.View и предоставляет им богатые возможности управления событиями. Обратите внимание на то, что традиционную роль представления (в стиле языка Smalltalk-80) играет шаблон, а не компонент Backbone.View.

На этом наше первое знакомство с фреймворком Backbone.js завершается. Ниже мы поговорим о его возможностях, основанных на приведенных здесь простых конструкциях. Перед тем как двигаться дальше, давайте познакомимся с ключевыми возможностями JavaScript-фреймворков на основе паттернов MV*.

Особенности реализации

Одностраничное приложение загружается в браузер с помощью обычного HTTP-запроса и HTTP-ответа. Страница может представлять собой HTML-файл, как в предыдущем примере, или являться представлением, сгенерированным серверной стороной реализации MVC.

Как только одностраничное приложение загружено, маршрутизатор клиентской стороны перехватывает URL и обращается к логике клиента вместо того,