РАЗРАБОТКА ВЕБ-САЙТОВ*, JAVASCRIPT*, БЛОГ КОМПАНИИ RUVDS.COM

MVC на чистом JavaScript

ПЕРЕВОД

ru_vds 21 июля 2017 в 15:00 21,4k

Оригинал: Camilo Reyes

Шаблоны проектирования часто встраивают в популярные фреймворки. Например, шаблон MVC (Model-View-Controller, Модель-Представление-Контроллер) можно встретить буквально повсюду. В JavaScript трудно отделить фреймворк от реализованного в нём шаблона проектирования, причём, часто авторы фреймворков интерпретируют MVC по-своему и навязывают программистам своё видение вопроса.

То, как именно будет выглядеть конкретная реализация MVC, полностью зависит от фреймворка. В результате мы получаем массу разных реализаций, что сбивает с толку и ведёт к беспорядку. Особенно это заметно, когда в одном проекте используется несколько фреймворков. Эта ситуация заставила меня задаться вопросом: «А есть ли способ лучше?».

Шаблон MVC хорош для клиентских фреймворков, однако, полагаясь на нечто «современное», нужно помнить о том, что уже завтра появится что-то новое, а то, что современно сегодня, устареет. Это — тоже проблема, и мне хотелось бы исследовать альтернативы фреймворкам и посмотреть, к чему всё это может привести.

Шаблон проектирования MVC появился несколько десятков лет

назад. Полагаю, в его изучение стоит вложить время любому программисту. Этот шаблон можно использовать без привязки к каким-либо фреймворкам.

Реализация MVC — это ещё один фреймворк?

Для начала мне хотелось бы развеять один распространённый миф, когда ставят знак равенства между шаблонами проектирования и фреймворками. Это — разные вещи.

Шаблон — это строгий подход к решению некоей задачи. Для реализации шаблона, за детали которой отвечает программист, нужен определённый уровень квалификации. Шаблон MVC позволяет следовать принципу разделения ответственностей и способствует написанию чистого кода.

Фреймворки не привязаны к конкретным шаблонам проектирования. Для того, чтобы отличить фреймворк от шаблона, можно воспользоваться так называемым голливудским принципом: «не звоните нам, мы сами вам позвоним». Если в системе имеется некая зависимость и при этом в определённой ситуации вы вынуждены её использовать — это фреймворк. Схожесть фреймворков с голливудом заключается в том, что разработчики, пользующиеся ими, похожи на актёров, которые вынуждены строго следовать сценариям фильмов. У таких программистов нет права голоса.

Стоит ли избегать клиентских фреймворков? Каждый сам ответит на этот вопрос, однако, вот несколько веских причин от них

отказаться:

- Фреймворки усложняют решения и увеличивают риск сбоев.
- Проект сильно зависит от фреймворка, что ведёт к появлению трудноподдерживаемого кода.
- При появлении новых версий фреймворков сложно с переписывать под них существующий код.

Шаблон MVC

Шаблон проектирования MVC родом из 1970-х. Он появился в научно-исследовательском центре Xerox PARC в ходе работы над языком программирования Smalltalk. Шаблон прошёл проверку временем в деле разработки графических пользовательских интерфейсов. Он пришёл в веб-программирование из настольных приложений и доказал свою эффективность в новой сфере применения.

По сути, MVC — это способ чёткого разделения ответственностей. В результате конструкция решения, основанного на нём, оказывается понятной даже новому программисту, который по каким-то причинам присоединился к проекту. Как результат, даже тому, кто с проектом знаком не был, легко в нём разобраться, и, при необходимости, внести вклад в его разработку.

Пример реализации MVC

Чтобы было веселей, напишем веб-приложение, которое будет посвящено пингвинам. Кстати, эти милые существа, похожие на

плюшевые игрушки, живут не только в антарктических льдах. Всего существует около полутора десятка видов пингвинов. Пришло время на них взглянуть. А именно, наше приложение будет состоять из одной веб-страницы, на которой расположена область просмотра сведений о пингвинах и пара кнопок, которые позволяют просматривать каталог пингвинов.

При создании приложения пользоваться мы будем шаблоном MVC, строго следуя его принципам. Кроме того, в процессе решения задачи будет задействована методология экстремального программирования, а также модульные тесты. Всё будет сделано на JS, HTML и CSS — никаких фреймворков, ничего лишнего.

Освоив этот пример, вы узнаете достаточно для того, чтобы интегрировать MVC в собственные проекты. Кроме того, так как ПО, построенное с использованием этого шаблона, отлично поддаётся тестированию, попутно вы ознакомитесь с модульными тестами для нашего учебного проекта. Понятное дело, если надо, их вы тоже сможете взять на вооружение.

Мы будем придерживаться стандарта ES5 для обеспечения кросс-браузерной совместимости. Полагаем, шаблон MVC вполне заслужил того, чтобы для его реализации использовались широко известные, проверенные возможности языка.

Итак, приступим.

Общий обзор проекта

Демонстрационный проект будет состоять из трёх основных компонентов: контроллера, представления и модели. Каждый из них имеет собственную сферу ответственности и ориентирован на решение конкретной задачи.

Вот как это выглядит в виде схемы.

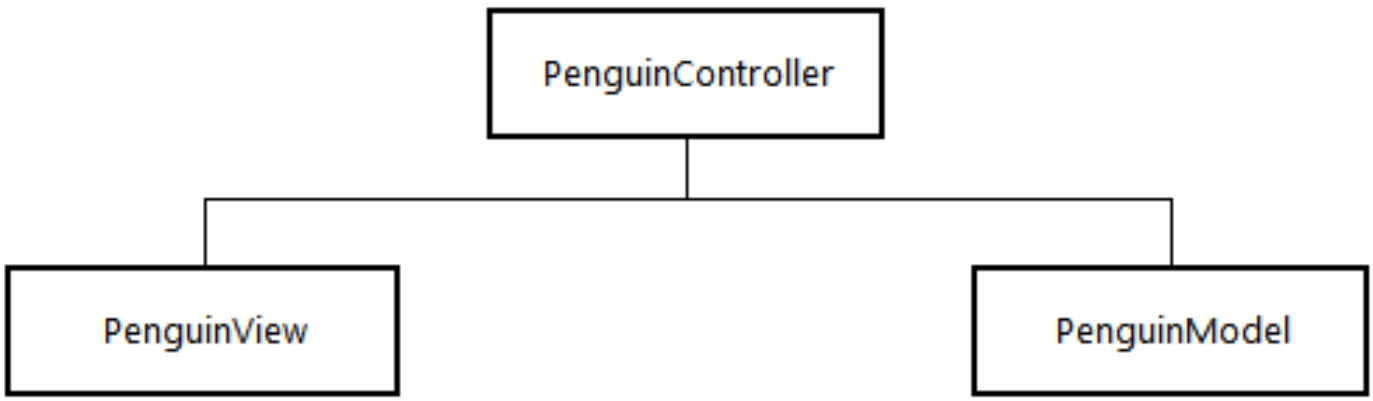


Схема проекта

Контроллер `PenguinController` занимается обработкой событий и служит посредником между представлением и моделью. Он выясняет, что произошло, когда пользователь выполняет некое действие (например, щёлкает по кнопке или нажимает клавишу на клавиатуре). Логика клиентских приложений может быть реализована в контроллере. В более крупных системах, в которых нужно обрабатывать множество событий, этот элемент можно разбить на несколько модулей. Контроллер является входной точкой для событий и единственным посредником между представлением и данными.

Представление `PenguinView` взаимодействует с DOM. DOM —

это API браузера, с помощью которого работают с HTML. В MVC только представление отвечает за изменения DOM. Представление может выполнять подключение обработчиков событий пользовательского интерфейса, но обработка событий — прерогатива контроллера. Основная задача, решаемая представлением — управлять тем, что пользователь видит на экране. В нашем проекте представление будет выполнять манипуляции с DOM, используя JavaScript.

Модель `PenguinModel` отвечает за работу с данными. В клиентском JS это означает выполнение Ajax-операций. Одно из преимуществ шаблона MVC заключается в том, что всё взаимодействие с источником данных, например — с сервером, сосредоточено в одном месте. Такой подход помогает программистам, которые не знакомы с проектом, разобраться в нём. Модель в этом шаблоне проектирования занята исключительно работой с JSON или объектами, которые поступают с сервера.

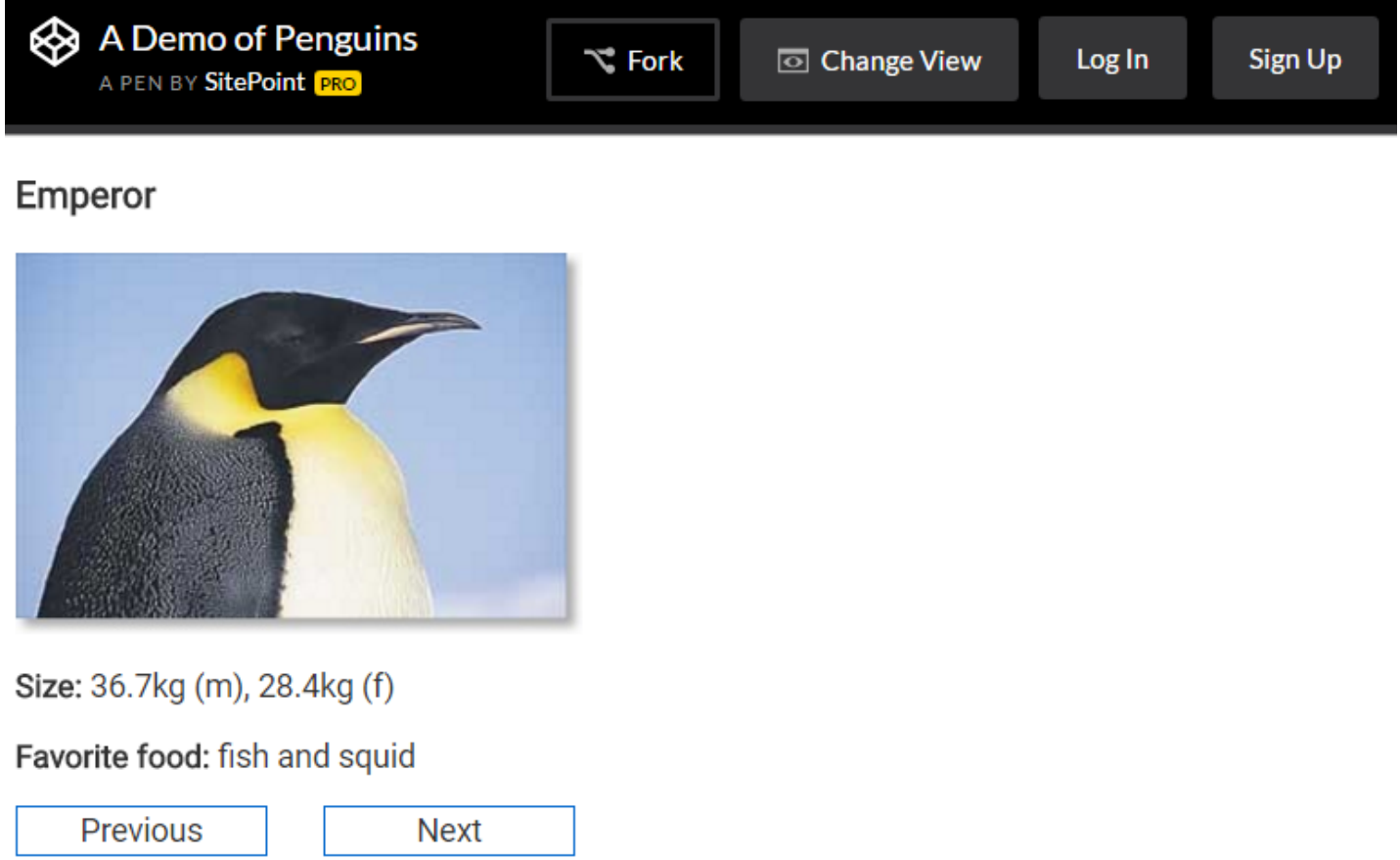
Если при реализации MVC нарушить вышеописанное разделение сфер ответственности компонентов, мы получим один из возможных анти-паттернов MVC. Модель не должна работать с HTML. Представление не должно выполнять Ajax-запросов. Контроллер должен играть роль посредника, не заботясь о деталях реализации других компонентов.

Если веб-разработчик, при реализации MVC, уделяет недостаточно внимания разделению ответственности компонентов, всё, в итоге, превращается в один веб-компонент. В результате,

несмотря на хорошие намерения, получается беспорядок. Это происходит из-за того, что повышенное внимание уделяется возможностям приложения и всему тому, что связано со взаимодействием с пользователем. Однако, разделение ответственности компонентов в сферах возможностей программы — это не то же самое, что разделение по функциям.

Полагаю, в программировании стоит стремиться к чёткому разделению функциональных сфер ответственности. В результате каждая задача оказывается обеспеченной единообразным способом её решения. Это делает код понятнее, позволяет новым программистам, подключающимся к проекту, быстрее понять код и приступить к продуктивной работе над ним.

Пожалуй, довольно рассуждений, пришло время взглянуть на рабочий пример, код которого размещён на CodePen. Можете поэкспериментировать с ним.



Приложение на CodePen

Рассмотрим этот код.

Контроллер

Контроллер взаимодействует с представлением и моделью. Компоненты, необходимые для работы контроллера, имеются в его конструкторе:

```
var PenguinController = function PenguinController(penguinView,
penguinModel) {
  this.penguinView = penguinView;
  this.penguinModel = penguinModel;
};
```

Конструктор использует инверсию управления, модули внедряются

в него в соответствии с этой идеей. Инверсия управления позволяет внедрять любые компоненты, соответствующие определённым высокоуровневым контрактам. Это можно рассматривать как удобный способ абстрагирования от деталей реализации. Подобный подход способствует написанию чистого кода на JavaScript.

Затем подключаются события, связанные со взаимодействием с пользователем:

```
PenguinController.prototype.initialize = function initialize() {
  this.penguinView.onClickGetPenguin =
this.onClickGetPenguin.bind(this);
};

PenguinController.prototype.onClickGetPenguin = function
onClickGetPenguin(e) {
  var target = e.currentTarget;
  var index = parseInt(target.dataset.penguinIndex, 10);

  this.penguinModel.getPenguin(index, this.showPenguin.bind(this));
};
```

Обратите внимание на то, что обработчики событий используют данные о состоянии приложения, хранящиеся в DOM. В данном случае этой информации нам достаточно. Текущее состояние DOM — это то, что пользователь видит в браузере. Данные о состоянии приложения можно хранить непосредственно в DOM, но контроллер не должен самостоятельно воздействовать на состояние приложения.

Когда происходит событие, контроллер считывает данные и принимает решения о дальнейших действиях. В данный момент

речь идёт о функции обратного вызова `this.showPenguin()`:

```
PenguinController.prototype.showPenguin = function
showPenguin(penguinModelData) {
  var penguinViewModel = {
    name: penguinModelData.name,
    imageUrl: penguinModelData.imageUrl,
    size: penguinModelData.size,
    favoriteFood: penguinModelData.favoriteFood
  };

  penguinViewModel.previousIndex = penguinModelData.index - 1;
  penguinViewModel.nextIndex = penguinModelData.index + 1;

  if (penguinModelData.index === 0) {
    penguinViewModel.previousIndex = penguinModelData.count - 1;
  }

  if (penguinModelData.index === penguinModelData.count - 1) {
    penguinViewModel.nextIndex = 0;
  }

  this.penguinView.render(penguinViewModel);
};
```

Контроллер, основываясь на состоянии приложения и на произошедшем событии, находит индекс, соответствующий набору данных о пингвине, и сообщает представлению о том, что именно нужно вывести на странице. Контроллер берёт необходимые данные из модели и преобразует их в объект, с которым может работать представление.

Представленные здесь модульные тесты построены по модели AAA (Arrange, Act, Assert — размещение, действие, утверждение). Вот модульный тест для стандартного сценария показа информации о пингвине:

```
var PenguinViewMock = function PenguinViewMock() {
  this.calledRenderWith = null;
};

PenguinViewMock.prototype.render = function render(penguinViewModel)
{
  this.calledRenderWith = penguinViewModel;
};

// Arrange
var penguinViewMock = new PenguinViewMock();

var controller = new PenguinController(penguinViewMock, null);

var penguinModelData = {
  name: 'Chinstrap',
  imageUrl: 'http://chinstrapl.jpg',
  size: '5.0kg (m), 4.8kg (f)',
  favoriteFood: 'krill',
  index: 2,
  count: 5
};

// Act
controller.showPenguin(penguinModelData);

// Assert
assert.strictEqual(penguinViewMock.calledRenderWith.name,
'Chinstrap');
assert.strictEqual(penguinViewMock.calledRenderWith.imageUrl,
'http://chinstrapl.jpg');
assert.strictEqual(penguinViewMock.calledRenderWith.size, '5.0kg
(m), 4.8kg (f)');
assert.strictEqual(penguinViewMock.calledRenderWith.favoriteFood,
'krill');
assert.strictEqual(penguinViewMock.calledRenderWith.previousIndex,
1);
assert.strictEqual(penguinViewMock.calledRenderWith.nextIndex, 3);
```

Объект-заглушка `PenguinViewMock` реализует тот же контракт, что и реальный модуль представления. Это позволяет писать модульные тесты и проверять, в блоке Assert, всё ли работает так, как нужно.

Объект `assert` взят из Node.js, но можно воспользоваться аналогичным объектом из библиотеки Chai. Это позволяет писать тесты, которые можно выполнять и на сервере, и в браузере.

Обратите внимание на то, что контроллер не заботится о деталях реализации. Он полагается на контракт, который предоставляет представление, вроде `this.render()`. Именно такого подхода необходимо придерживаться для написания чистого кода. Контроллер, при таком подходе, может доверить компоненту выполнение тех задач, о возможности выполнения которых заявил этот компонент. Это делает структуру проекта прозрачной, что улучшает читаемость кода.

Представление

Представление заботится лишь об элементах DOM и о подключении обработчиков событий. Например:

```
var PenguinView = function PenguinView(element) {
  this.element = element;

  this.onClickGetPenguin = null;
};
```

Вот как реализуется в коде воздействие представления на то, что видит пользователь:

```
PenguinView.prototype.render = function render(viewModel) {
  this.element.innerHTML = '<h3>' + viewModel.name + '</h3>' +
    '<img class="penguin-image" src="' + viewModel.imageUrl +
```

```
      '" alt="' + viewModel.name + '" />' +
    '<p><b>Size:</b> ' + viewModel.size + '</p>' +
    '<p><b>Favorite food:</b> ' + viewModel.favoriteFood + '</p>' +
    '<a id="previousPenguin" class="previous button"
href="javascript:void(0);"' +
      ' data-penguin-index="' + viewModel.previousIndex +
'">Previous</a> ' +
    '<a id="nextPenguin" class="next button"
href="javascript:void(0);"' +
      ' data-penguin-index="' + viewModel.nextIndex + '">Next</a>';

  this.previousIndex = viewModel.previousIndex;
  this.nextIndex = viewModel.nextIndex;

  // Подключение обработчиков событий щелчков по кнопкам и передача
задачи обработки событий контроллеру
  var previousPenguin =
this.element.querySelector('#previousPenguin');
  previousPenguin.addEventListener('click', this.onClickGetPenguin);

  var nextPenguin = this.element.querySelector('#nextPenguin');
  nextPenguin.addEventListener('click', this.onClickGetPenguin);
  nextPenguin.focus();
}
```

Обратите внимание на то, что основная задача представления заключается в том, чтобы превратить данные, полученные из модели, в HTML, и поменять состояние приложения. Ещё одна задача — подключение обработчиков событий и передача функций их обработки контроллеру Обработчики событий подключаются к DOM после изменения состояния. Этот подход позволяет просто и удобно управлять событиями.

Для того, чтобы всё это протестировать, мы можем проверить обновление элементов и изменение состояния приложения:

```
var ElementMock = function ElementMock() {
  this.innerHTML = null;
};
```

```
// Функции-заглушки, необходимые для того, чтобы провести
тестирование
ElementMock.prototype.querySelector = function querySelector() { };
ElementMock.prototype.addEventListener = function addEventListener()
{ };
ElementMock.prototype.focus = function focus() { };

// Arrange
var elementMock = new ElementMock();

var view = new PenguinView(elementMock);

var viewModel = {
  name: 'Chinstrap',
  imageUrl: 'http://chinstrap1.jpg',
  size: '5.0kg (m), 4.8kg (f)',
  favoriteFood: 'krill',
  previousIndex: 1,
  nextIndex: 2
};

// Act
view.render(viewModel);

// Assert
assert(elementMock.innerHTML.indexOf(viewModel.name) > 0);
assert(elementMock.innerHTML.indexOf(viewModel.imageUrl) > 0);
assert(elementMock.innerHTML.indexOf(viewModel.size) > 0);
assert(elementMock.innerHTML.indexOf(viewModel.favoriteFood) > 0);
assert(elementMock.innerHTML.indexOf(viewModel.previousIndex) > 0);
assert(elementMock.innerHTML.indexOf(viewModel.nextIndex) > 0);
```

Мы рассмотрели представление и контроллер. Они решают большую часть задач, возлагаемых на приложение. А именно, отвечают за то, что видит пользователь и позволяют ему взаимодействовать с программой через механизм событий. Осталось разобраться с тем, откуда берутся данные, которые выводятся на экран.

Модель

В шаблоне MVC модель занята взаимодействием с источником данных. В нашем случае — с севером. Например:

```
var PenguinModel = function PenguinModel(XMLHttpRequest) {
  this.XMLHttpRequest = XMLHttpRequest;
};
```

Обратите внимание на то, что модуль `XMLHttpRequest` внедрён в конструктор модели. Это, кроме прочего, подсказка для других программистов касательно компонентов, необходимых модели. Если модель нуждается в различных способах работы с данными, в неё можно внедрить и другие модули. Так же, как и в рассмотренных выше случаях, для модели можно подготовить модульные тесты.

Получим данные о пингвине, основываясь на индексе:

```
PenguinModel.prototype.getPenguin = function getPenguin(index, fn) {
  var oReq = new this.XMLHttpRequest();

  oReq.onload = function onLoad(e) {
    var ajaxResponse = JSON.parse(e.currentTarget.responseText);
    // Индекс должен быть целым числом, иначе это работать не будет
    var penguin = ajaxResponse[index];

    penguin.index = index;
    penguin.count = ajaxResponse.length;

    fn(penguin);
  };

  oReq.open('GET',
'https://codepen.io/beautifulcoder/pen/vmOOLr.js', true);
  oReq.send();
};
```

Тут осуществляется подключение к серверу и загрузка с него данных. Проверим компонент с помощью модульного теста и условных тестовых данных:

```
var LIST_OF_PENGUINS =
'[{"name":"Emperor","imageUrl":"http://imageUrl",' +
  '"size":"36.7kg (m), 28.4kg (f)","favoriteFood":"fish and
squid"}]';

var XMLHttpRequestMock = function XMLHttpRequestMock() {
  // Для целей тестирования нужно это установить, иначе тест не
удастся
  this.onload = null;
};

XMLHttpRequestMock.prototype.open = function open(method, url,
async) {
  // Внутренние проверки, система должна иметь конечные точки method
и url
  assert(method);
  assert(url);
  // Если Ajax не асинхронен, значит наша реализация весьма неудачна
:-)
  assert.strictEqual(async, true);
};

XMLHttpRequestMock.prototype.send = function send() {
  // Функция обратного вызова симулирует Ajax-запрос
  this.onload({ currentTarget: { responseText: LIST_OF_PENGUINS }
});
};

// Arrange
var penguinModel = new PenguinModel(XMLHttpRequestMock);

// Act
penguinModel.getPenguin(0, function onPenguinData(penguinData) {

  // Assert
  assert.strictEqual(penguinData.name, 'Emperor');
  assert(penguinData.imageUrl);
  assert.strictEqual(penguinData.size, '36.7kg (m), 28.4kg (f)');
  assert.strictEqual(penguinData.favoriteFood, 'fish and squid');
```

```
  assert.strictEqual(penguinData.index, 0);
  assert.strictEqual(penguinData.count, 1);
});
```

Как видите, модель заботят лишь необработанные данные. Это означает работу с Ajax и с JavaScript-объектами. Если вы не вполне владеете темой Ajax в JavaScript, вот полезный материал об этом.

Модульные тесты

При любых правилах написания кода важно выполнять проверку того, что получилось. Шаблон проектирования MVC не регламентирует способ решения задачи. В рамках шаблона очерчены границы, довольно свободные, не пересекая которых можно писать чистый код. Это даёт свободу от засилья зависимостей.

Я обычно стремлюсь к тому, чтобы иметь полный комплект модульных тестов, охватывающих каждый вариант использования продукта. Эти тесты можно рассматривать как рекомендации по использованию кода. Подобный подход делает проект более открытым, понятным для любого программиста, желающего поучаствовать в его развитии.

Поэкспериментируйте с полным набором модульных тестов. Они помогут вам лучше понять описываемый здесь шаблон проектирования. Каждый тест предназначен для проверки конкретного варианта использования программы. Модульные тесты помогут чётко разделять задачи, и, при реализации того или

иного функционал проекта, не отвлекаться на другие его части.

О развитии учебного проекта

Наш проект реализует лишь базовый функционал, необходимый для того, чтобы продемонстрировать реализацию MVC. Однако, его можно расширить. Например, вот некоторые из возможных улучшений:

- Добавить экран со списком всех пингвинов.
- Добавить обработку событий клавиатуры для организации альтернативного способа переключения между карточками пингвинов. Аналогично, можно, для мобильных устройств, добавить управление жестами на сенсорном экране.
- Добавить SVG-диаграмму для визуализации данных. Например, так можно вывести обобщённые сведения о размерах пингвинов.

Если вы хотите, в качестве упражнения, развить этот проект, учитывайте, что выше приведены лишь несколько идей. Вы вполне можете, с использованием MVC, реализовать и другой функционал.

Итоги

Надеемся, вы оценили преимущества, которые дают шаблон проектирования MVC и дисциплинированный подход к архитектуре и коду проекта. Хороший шаблон проектирования, с одной стороны, не мешает работе, с другой, способствует написанию

чистого кода. Он, в процессе решения задачи, не даёт вам свернуть с правильного пути. Это повышает эффективность работы программиста.

Программирование — это искусство последовательного решения задач, разбиения системы на небольшие части, реализующие определённый функционал. В MVC это означает строгое следование принципу разделения сфер функциональной ответственности компонентов.

Разработчики обычно считают, что они не поддаются действию эмоций, что их действия всегда логичны. Однако, при попытке одновременно решать несколько задач, человек оказывается в тяжёлой психологической ситуации. Нагрузка оказывается слишком высокой. Работа в подобном состоянии плохо влияет на качество кода. В определённый момент логика отходит на второй план, как результат, растёт риск возникновения ошибок, ухудшается качество кода.

Именно поэтому рекомендуется разбивать проекты на небольшие задачи и решать их по очереди. Дисциплинированный подход к реализации MVC и подготовка модульных тестов для различных частей системы способствуют спокойной и продуктивной работе.

Уважаемые читатели! Какие шаблоны проектирования вы применяете в своих JS-проектах?

Поделиться:

Сохранить:

Комментарии (28)

Похожие публикации

JavaScript-прокси: и красиво, и полезно

13

ПЕРЕВОД

ru_vds • 29 января в 12:23

Может ли в JavaScript конструкция (a==1 && a==2 && a==3) оказаться равной true?

95

ПЕРЕВОД

ru_vds • 25 января в 16:08

JavaScript и ужасы мутаций

48

ПЕРЕВОД

ru_vds • 19 января в 11:42

Популярное за сутки

Наташа — библиотека для извлечения структурированной информации из текстов на русском языке

14

alexkuku • вчера в 16:12

Unit-тестирование скриншотами: преодолеваем звуковой барьер. Расшифровка доклада

4

lahmatiy • вчера в 13:05

Люди не хотят чего-то действительно нового — они хотят привычное, но сделанное иначе

25

ПЕРЕВОД

Smileek • вчера в 10:32

Руководство по SEO JavaScript-сайтов. Часть 2. Проблемы, эксперименты и рекомендации

2

ПЕРЕВОД

ru_vds • вчера в 12:04

Как адаптировать игру на Unity под iPhone X к апрелю

0

P1CACHU • вчера в 16:13

Лучшее на Geektimes

Стивен Хокинг, автор «Краткой истории времени», умер на 77 году жизни

33

HostingManager • вчера в 13:49

Обзор рынка моноколес 2018

70

lozga • вчера в 06:58

«Битва за Telegram»: 35 пользователей подали в суд на ФСБ

40

alizar • вчера в 15:14

Стивен Хокинг и его работа — что дал ученый человечеству?

8

marks • вчера в 14:46

Sunlike — светодиодный свет нового поколения

17

AlexeyNadezhin • вчера в 20:32