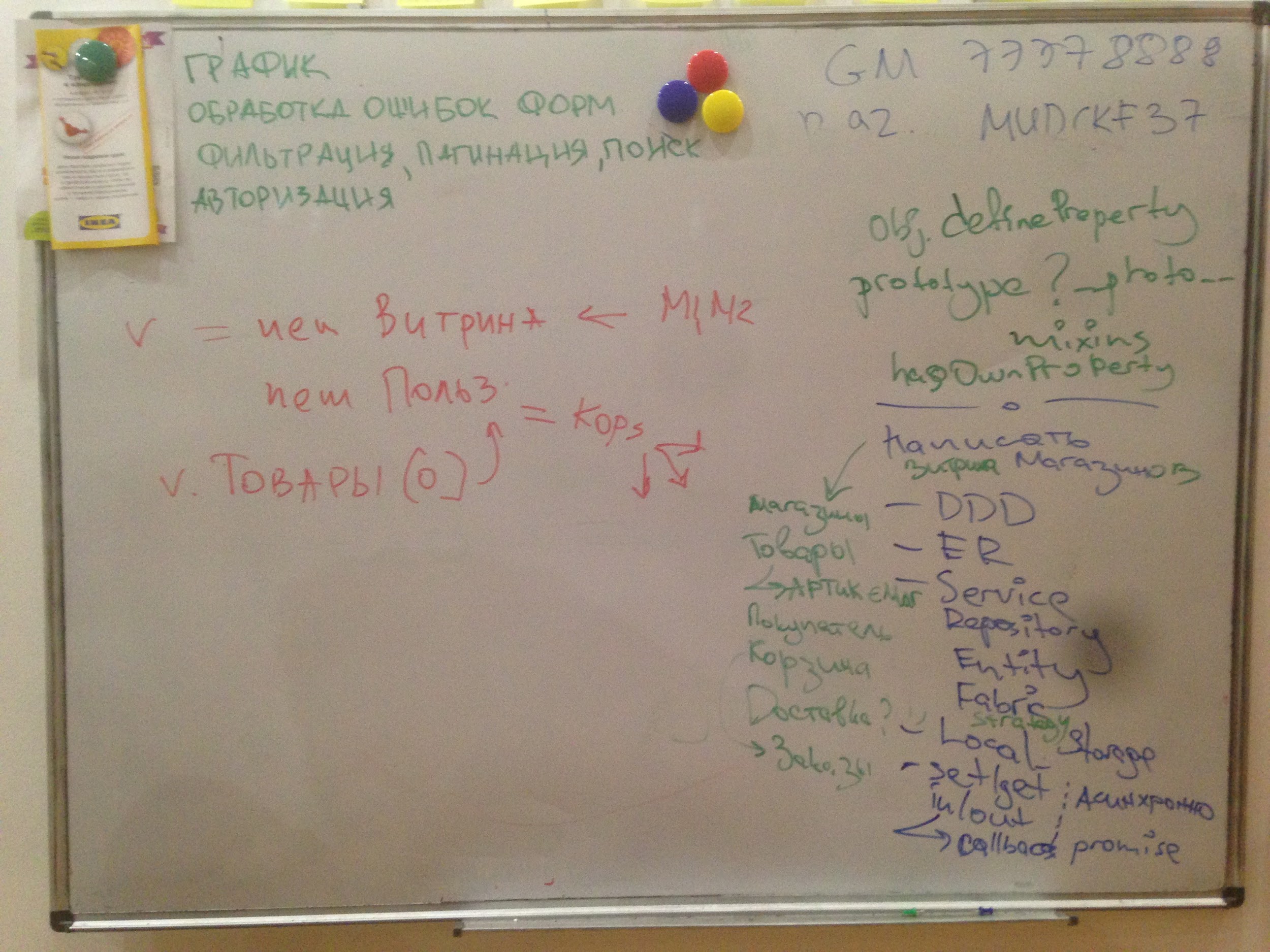
1. Базовые управляющие структуры
2. Функции
3. Наследование. Наследование через примеси. Прототипы. Прототипное наследование (Прототип, как реализация паттерна “Прототип”)

Проект: EMarket.

Описание:



repo:

v0.4:

<http://habrahabr.ru/post/217901/> - node.js exports, require

Добавляем нормальные тесты (jasmine)

<http://jasmine.github.io/> - ссылка на framework

<http://habrahabr.ru/post/167173/> - Введение в Jasmine

<http://habrahabr.ru/post/169699/> - Jasmine - доп. возможности (продолжение введения)

<https://github.com/jasmine/jasmine> - установка Jasmine

<http://jasmine.github.io/2.0/introduction.html> - official introduction to Jasmine

<http://habrahabr.ru/post/207794/> - статья о том, как писать и не пиcать тесты

Code review tool: JSLint, JSHint

<http://jshint.com/install/> - переходим на JSHint

можно установить по-разному:

* npm
* sublime plugin (JSHint Gutter)

config файл:

{

"jasmine": true,

"strict": false,

"node": true

}

еще один пример config-файла:

<https://gist.github.com/connor/1597131>

Ставим karma

<http://karma-runner.github.io/0.10/>

<http://karma-runner.github.io/0.10/config/configuration-file.html> - configuring karma

<http://habrahabr.ru/post/187094/> - установка и запуск кармы

*не получилось :(*

Возвращаемся к Javascript:

<http://habrahabr.ru/company/enterra/blog/153365/> - ООП в JS

<http://javascript.ru/tutorial/object/inheritance> - ООП в JS - способы реализации

Прописовав своими силами модельки и прочее, прихожу к выводу, что надо использовать готовые и зарекомендовавшие себя решения, как минимум, чтобы посмотреть, как это сделано там.

Смотрим на backbone:

<http://habrahabr.ru/post/127049/>

<http://backbonejs.org/>

<http://nodeguide.ru/doc/dailyjs-backbone/backbone-tutorial-1/>

underscore:

<http://habrahabr.ru/post/240713/>

<http://underscorejs.org/docs/underscore.html>

State machine on JS

<https://github.com/jakesgordon/javascript-state-machine>

<http://machina-js.org/>

Breeze:

<http://breeze.github.io/>

крутая штука типа ORM

CQRS on JS

Command-query separation (CQS) или command-query responsibility segregation (CQRS) — это принцип [императивного программирования](https://ru.wikipedia.org/wiki/%D0%98%D0%BC%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), изобретенный [Бертраном Мейером](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D0%B9%D0%B5%D1%80,_%D0%91%D0%B5%D1%80%D1%82%D1%80%D0%B0%D0%BD) во время работы над языком программирования [Eiffel](https://ru.wikipedia.org/wiki/Eiffel).

Принцип гласит, что метод должен быть либо командой, выполняющей какое-то действие, либо запросом, возвращающим данные, но не одновременно. Другими словами, задавание вопроса не должно менять ответ. Более формально, возвращать значение можно только [чистым](https://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D1%82%D0%BE%D1%82%D0%B0_%D1%8F%D0%B7%D1%8B%D0%BA%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), не имеющим [побочных эффектов](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B1%D0%BE%D1%87%D0%BD%D1%8B%D0%B9_%D1%8D%D1%84%D1%84%D0%B5%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) методам. Следует отметить, что неукоснительное соблюдение этого принципа делает невозможным отслеживание количества вызовов запросов.

<http://www.udidahan.com/2009/12/09/clarified-cqrs/>

<http://cqrs.js.org/pages/domain.html>

Refactoring JS

Patterns: <http://journal.crushlovely.com/post/88286828068/7-patterns-to-refactor-javascript-applications>

Bryan’s article describes Value Objects as “…simple objects whose equality is dependent on their value rather than an identity.”

Service Objects are objects that perform a discrete operation or action. When a process becomes complex, hard to test, or touches more than one type of model, a Service Object can come in handy for cleaning up your code base.

Forms often have complex logic applied to them. In general, the logic breaks down into the following categories: validation, persistence or other operations, and feedback

A Form Object can encapsulate all associated logic into a single object, keeping it focused, isolated, and easy to test.

Query objects provide a nice tool for extracting query logic and associated operations into a contained module, pulling the logic out into a more maintainable and readable structure, while also providing a very readable API where the query object is used.

Storing view-specific attributes on the model can create confusion about what is “truth” (stored in the database) and what is purely representational. View Objects act as a kind of adapter between the truth and the representation of the truth.

What View Objects do is “dress up” the data, by way of transforming, adding or removing data properties, returning a new object for use in the presentation layer. This approach creates a nice home for our presentation-specific logic and attributes, keeping it removed from the model truth.

When a piece of business logic associated with a model becomes sufficiently complex or is not a part of the core model logic, it is a candidate for extraction into a Policy Object. These objects encapsulate operations that interpret models and exclusively return boolean values, describing whether the policy passes or doesn’t pass the object.

Policy Objects are similar to Service Objects, but I use the term ‘Service Object’ for write operations and 'Policy Object’ for reads. They are also similar to Query Objects, but Query Objects focus on executing SQL to return a result set, whereas Policy Objects operate on domain models already loaded into memory.

When a process has side effects that need to be executed only in certain situations, this functionality can be layered onto an existing operation using Decorators. A Decorator takes an object and wraps auxiliary functionality around it, letting you add on what you need when you need it and keeping the core procedure untouched.

<http://jobs.crushlovely.com/>

Замыкания в JS

Замыкание - есть сохранение области видимости функции.

<http://habrahabr.ru/post/38642/>

<http://javascript.ru/basic/closure>

<https://learn.javascript.ru/functions-closures>

Функциональное программирование

<http://scott.sauyet.com/Javascript/Talk/FunctionalProgramming/>

Reg Braithwaite has a [good description](http://raganwald.com/2013/04/08/functional-vs-OOP.html) of the central difference between these two paradigms. OO focuses on the differences in the data, while FP concentrates on consistent data structures.

One main distinguishing characteristics of functional programming languages is that they describe what they want done, and not how to do it. OO, inside its methods, still uses mostly imperative techniques.

Documenting JS

<http://usejsdoc.org/about-getting-started.html>

**Тестовое**

1. Площадка для интернет-магазинов.

Роли:

* продавец
* покупатель

Продавец может:

* завести интернет-магазин
* выставить товары на продажу

Покупатель может:

* посмотреть существующие на площадке магазины
* посмотреть товары в конкретном магазине
* посмотреть “Витрину” - список всех товаров на площадке (общий список товаров со всех магазинов)
* выбрать товары, которые хочет приобрести (набрать в “Корзину”)
* оформить заказ и произвести списание со своего счета

В контексте данной задачи в первую очередь интересует именно программная оболочка, а не пользовательский интерфейс. То есть, достаточно будет сделать следующий (либо подобный) рабочий код: app.js:

|  |
| --- |
| var shop1 = Shop.create({name: "Shop1", city: "Kazan"});  var shop2 = Shop.create({name: "Shop2", city: "Moscow"});  var shop3 = Shop.create({name: "Shop3", city: "Kazan"});  var shop4 = Shop.create({name: "Shop4", city: "Moscow"});  shop1.addProduct({name: "T-Shirt", price: 100, amount: 50});  shop1.addProduct({name: "Cool T-Shirt", price: 200, amount: 50});  shop2.addProduct({name: "T-Shirt", price: 150, amount: 10});  shop3.addProduct({name: "iPhone 6", price: 50000, amount: 50});  var shops = Shop.getAll();  var shopsInKazan = Shop.findByCity("Kazan");  var products = Product.getAll();  var tshirts = Product.find("name", "t-shirt");  shop3.name = "The Best Shop";  shop3 = Shop.update(shop3);  var client = Client.create({name: "John Smith", city: "Kazan"});  client.refund(1000); // пополнить денежный счет  var product = tshirts[0];  client.addToOrder(product);  client.purchase();  client.addToOrder(tshirts[0]);  client.getOrders(); |

Для решения данной задачи предлагается использовать голый Javascript. Если есть соответствующие знания - покрыть данный код тестами.

Если не получается решить всю задачу целиком, то стоит сделать хотя бы часть, но чтобы она работала.

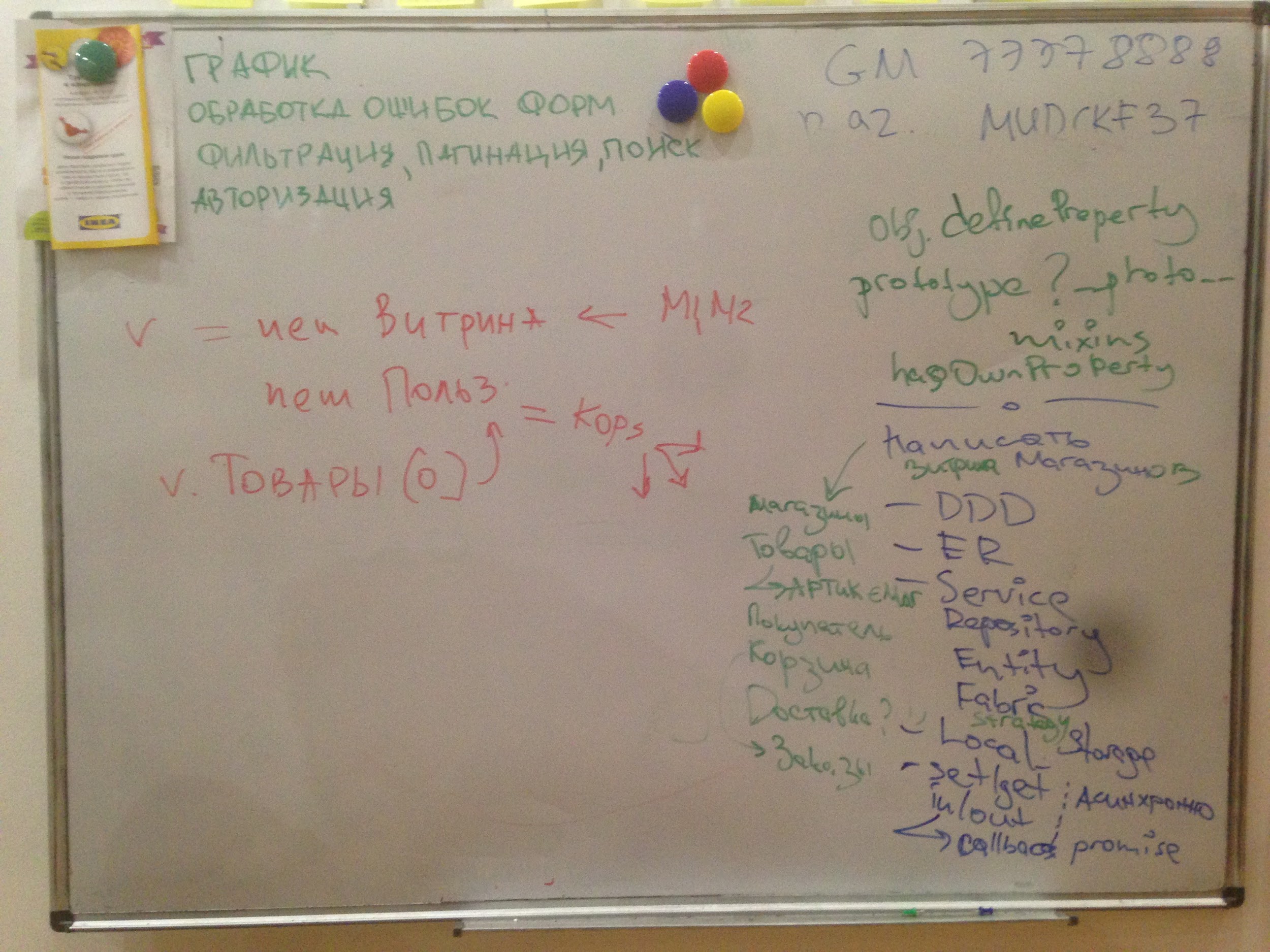
В роли хранилища предлагается использовать один из следующих вариантов:

* in-memory storage
* Local Storage (browser, HTML5 WebApi)
* CouchDB
* Что-либо другое по предварительному согласованию

В идеале решение предполагает использование DDD и различных паттернов проектирования по назначению, в частности:

* Service, Fabric, Decorator
* Repository, Storage
* Entity, ValueObject
* DataMapper/Active Record

Возможно не все перечисленные паттерны будут использованы, возможно, будут использованы и другие. Важно понимание и умение объяснить для чего нужен каждый из них + пример использования.



Материалы для изучения:

Книги:

[Крокфорд - JavaScript. Сильные стороны.pdf](https://vk.com/doc10903696_272054170?hash=25ef73bb275dc4cb0e&dl=5e6882daeafa9e6d56)

Секреты JavaScript ниндзя

Статьи:

<http://journal.crushlovely.com/post/88286828068/7-patterns-to-refactor-javascript-applications>

<http://habrahabr.ru/company/hexlet/blog/266443/>

<http://www.toptal.com/javascript/interview-questions>

<http://es5.javascript.ru/index.html>

Инструменты:

Language: Javascript

Code review tool: JSLint, JSHint

Platform: node.js

TDD: Karma, Jasmine, Browserify

Build&Deploy: Gulp, Webpack

Libs: Underscore

Documenting: JSDoc

Concepts

* Типы данных в JS
* Объекты в JS
  + Прототипы
* Функции в JS
  + Вызов функции
  + Область видимости
  + Замыкания
  + Обратные вызовы (callbacks)
  + Каррирование
* Наследование и ООП в JS
* HTTP protocol
* REST API

1. Перечислите типы данных.

2. Что такое встроенные объекты (built-in objects)?

3. Дайте определение замыканию.

4. В чем особенность области видимости в JS? Отличие от других языков.

5. Почему код выполнится без ошибок?

fn();

function fn() {}

6. Результат исполнения 0.1 + 0.2 = ? (особенность стандарта IEEE 754)

7. Что такое контекст выполнения функции?

8. Какой контескт при вызове fn и a.fn:

var fn = function () {};

var a = { fn: fn };

9. Как изменить контекст выполнения? А дважды?

10. Что такое arguments и какому типу он принадлежит?

11. Что такое строгий режим(strict) и в чем его особенность?

12. Чем отличаются свойства prototype и \_\_proto\_\_? Есть ли у них зависимость друг от друга?

13. Как создать объект у которого прототипом будет

var a = { foo: function () {} };

14. Как проверить является ли один объект прототипом другого объекта?

15. Как реализована абстракция "класс" в JS? Как создать экзепляр класса?

16. Как сделать чтобы функция выполнилась асинхронно?

17. В каком порядке выполняются функции отправленные на асинхронное исполнение?

18. Как прервать все таймеры?

19. Что выведется в консоль? Почему?

var obj = {

a: 1

};

(function(obj) {

obj = {

a: 2

};

})(obj);

console.log(obj.a);

20. Что выведется в консоль? Как исправить?

for (var i = 0; i < 10; i++) {

setTimeout(function () {

console.log(i);

}, 0);

}