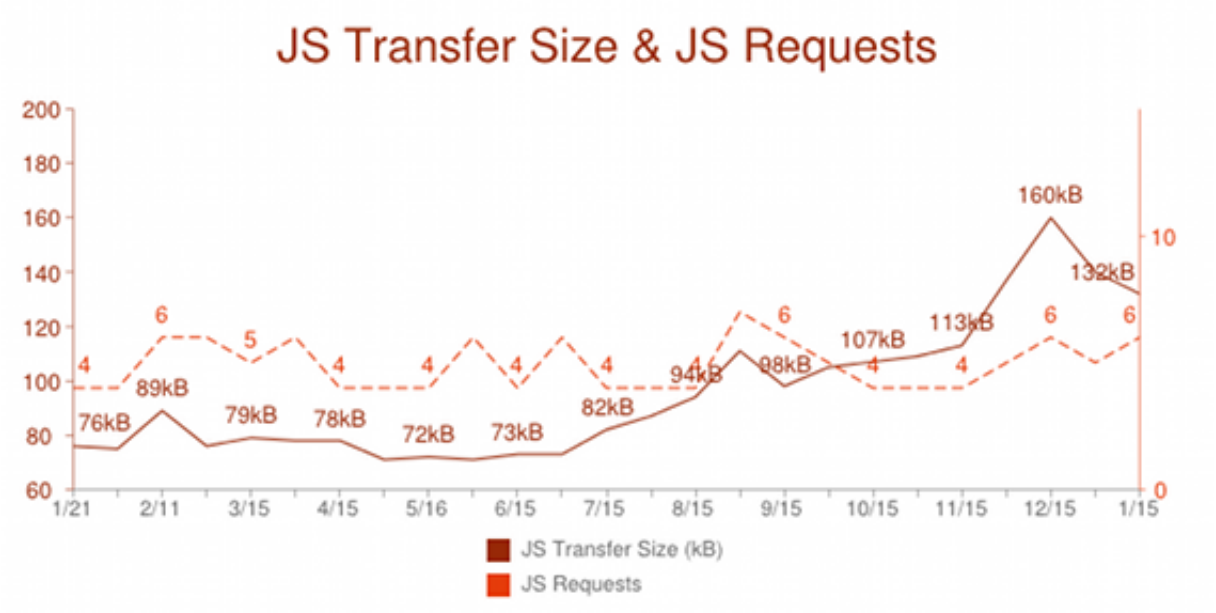
MVC Architecture

As modern browsers become more powerful with rich features, building full-blown web applications in JavaScript is not only feasible, but increasingly popular. Based on **trends** on **HTTP Archive**, deployed JavaScript code size has grown 45% over the course of the year.



With JavaScript's popularity climbing, our client-side applications are much more complex than before. Application development requires collaboration from multiple developers. Writing **maintainable** and **reusable** code is crucial in the new web app era. The Chrome App, with its rich client-side features, is no exception.

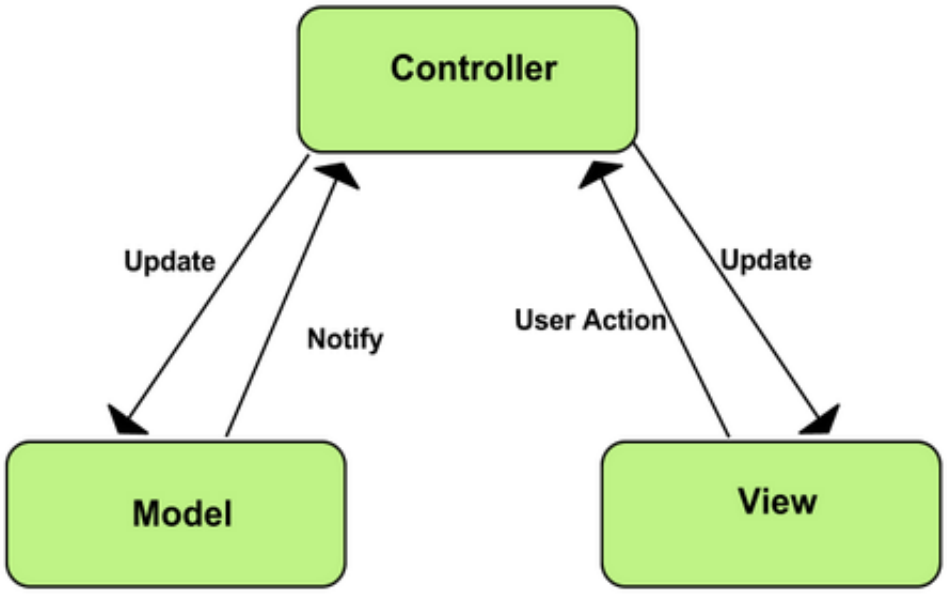
Design patterns are important to write maintainable and reusable code. A pattern is a reusable solution that can be applied to commonly occurring problems in software design — in our case — writing Chrome Apps. We recommend that developers decouple the app into a series of independent components following the MVC pattern.

In the last few years, a series of JavaScript MVC frameworks have been developed, such as **backbone.js**, **ember.js**, **AngularJS**, **Sencha**, **Kendo UI**, and more. While they all have their unique advantages, each one of them follows some form of MVC pattern with the goal of encouraging developers to write more structured JavaScript code.

MVC pattern overview

MVC offers architectural benefits over standard JavaScript — it helps you write better organized, and therefore more maintainable code. This pattern has been used and extensively tested over multiple languages and generations of programmers.

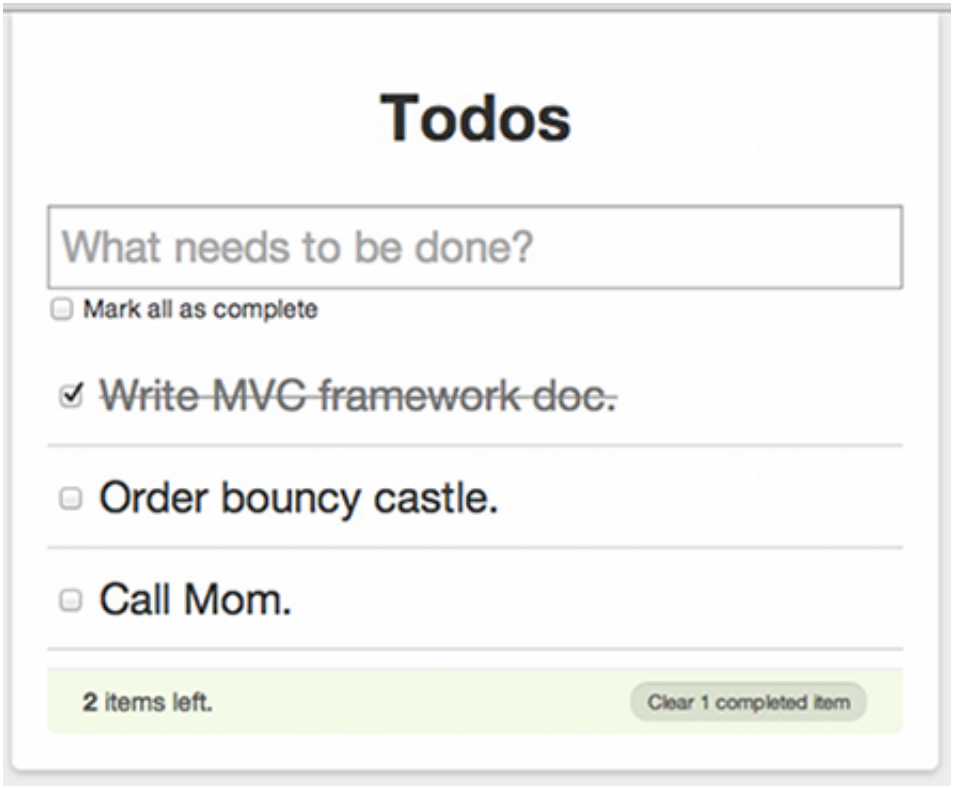
MVC is composed of three components:



Model

Model is where the application's data objects are stored. The model doesn't know anything about views and controllers. When a model changes, typically it will notify its observers that a change has occurred.

To understand this further, let's use the Todo list app, a simple, one page web app that tracks your task list.



The model here represents attributes associated with each todo item such as description and status. When a new todo item is created, it is stored in an instance of the model.

View

View is what's presented to the users and how users interact with the app. The view is made with HTML, CSS, JavaScript and often templates. This part of your Chrome App has access to the DOM.

For example, in the above todo list web app, you can create a view that nicely presents the list of todo items to your users. Users can also enter a new todo item through some input format; however, the view doesn't know how to update the model because that's the controller's job.

Controller

The controller is the decision maker and the glue between the model and view. The controller updates the view when the model changes. It also adds event listeners to the view and updates the model when the user manipulates the view.

In the todo list web app, when the user checks an item as completed, the click is forwarded to the controller. The controller modifies the model to mark item as completed. If the data needs to be persistent, it also makes an async save to the server. In rich client-side web app development such as Chrome Apps, keeping the data persistent in local storage is also crucial. In this case, the controller also handles saving the data to the client-side storage such as **FileSystem API**.

There are a few variations of the MVC design pattern such as MVP (Model–View–Presenter) and MVVP(Model–View–ViewModel). Even with the so called MVC design pattern itself, there is some variation between the traditional MVC pattern vs the modern interpretation in various programming languages. For example, some MVC–based frameworks will have the view observe the changes in the models while others will let the controller handle the view update. This article is not focused on the comparison of various implementations but rather on the separation–of–concerns and it's importance in writing modern web apps.

If you are interested in learning more, we recommend **Addy Osmani's** online book: **Learning JavaScript Design Patterns**.

To summarize, the MVC pattern brings modularity to application developers and it enables:

- Reusable and extendable code.
- Separation of view logic from business logic.
- Allow simultaneous work between developers who are responsible for different components (such

as UI layer and core logic).
- Easier to maintain.

MVC persistence patterns

There are many different ways of implementing persistence with an MVC framework, each with different trade–offs. When writing Chrome Apps, choose the frameworks with MVC and persistence patterns that feel natural to you and fit you application needs.

Model does its own persistence - ActiveRecord pattern

Popular in both server–side frameworks like Ruby on Rails, and client-side frameworks like **Backbone.js** and **ember.js**, the ActiveRecord pattern places the responsibility for persistence on the model itself and is typically implemented via JSON API.

A slightly different take from having a model handle the persistence is to introduce a separate concept of Store and Adapter API. Store, Model and Adapter (in some frameworks it is called Proxy) work hand by hand. Store is the repository that holds the loaded models, and it also provides functions such as creating, querying and filtering the model instances contained within it.

An adapter, or a proxy, receives the requests from a store and translates them into appropriate actions to take against your persistent data layer (such as JSON API). This is interesting in the modern web app design because you often interact with more than one persistent data layer such as a remote server and browser's local storage. Chrome Apps provides both **Chrome Storage API** and **HTML 5 fileSystem API** for client side storage.

Pros:

- Simple to use and understand.

Cons:

- Hard to test since the persistence layer is 'baked' into the object hierarchy.
- Having different objects use different persistent stores is difficult (for example, FileSystem APIs vs indexedDB vs server–side).
- Reusing Model in other applications may create conflicts, such as sharing a single Customer class between two different views, each view wanting to save to different places.

Controller does persistence

In this pattern, the controller holds a reference to both the model and a datastore and is responsible for keeping the model persisted. The controller responds to lifecycle events like Load, Save, Delete, and issues commands to the datastore to fetch or update the model.

Pros:

- Easier to test, controller can be passed a mock datastore to write tests against.
- The same model can be reused with multiple datastores just by constructing controllers with different datastores.

Cons:

- Code can be more complex to maintain.

AppController does persistence

In some patterns, there is a supervising controller responsible for navigating between one MVC and another. The AppController decides, for example, that a 'Back' button moves the client from an editing screen (which contains MVC widgets/formats), to a settings screen.

In the AppController pattern, the AppController responds to events and changes the app's current screen by issuing a call to the datastore to load any models needed and constructing all of the matching views and controllers for that screen.

Pros:

- Moves persistence layer even higher up the stack where it can be easily changed.
- Doesn't pollute lower level controllers like a DatePickerController with the need to know about persistence.

Cons:

- Each 'Page/Screen' of the app now requires a lot of boilerplate to write or update: Model, View, Controller, AppController.

Recommended MVC frameworks

MVC is crucial to designing Chrome Apps. We recommend the following **CSP–Compliant** MVC frameworks for writing secure and scalable Chrome Apps:

- **AngularJS** (**Text Drive Reference App** and **Build Apps with AngularJS tutorial**)

- **Kendo UI** (**Photo Booth Reference App**)
- **Sencha** (**Video Player Reference App** and **Build Apps with Sencha Ext JS tutorial**)

Useful resources

Online

- **HTML5Rocks.com**
- **Learning JavaScript Design Patterns** (by Addy Osmani)
- **TodoMVC**

Books

- **JavaScript Web Applications** (By Alex MacCaw)
- **JavaScript Patterns** (By Stoyan Stefonov)
- **Maintainable JavaScript** (By Nicolas Z. Zakas)

Back to top