EMBER.JS

A framework for creating
ambitious web applications

WHAT IS EMBER?

DATA BINDING.

COMPUTED PROPERTIES.

DATA-BOUND DECLARATIVE TEMPLATES.

We want it to be possible to define your view structure declaratively, with its data requirements, and leave code for handling events.

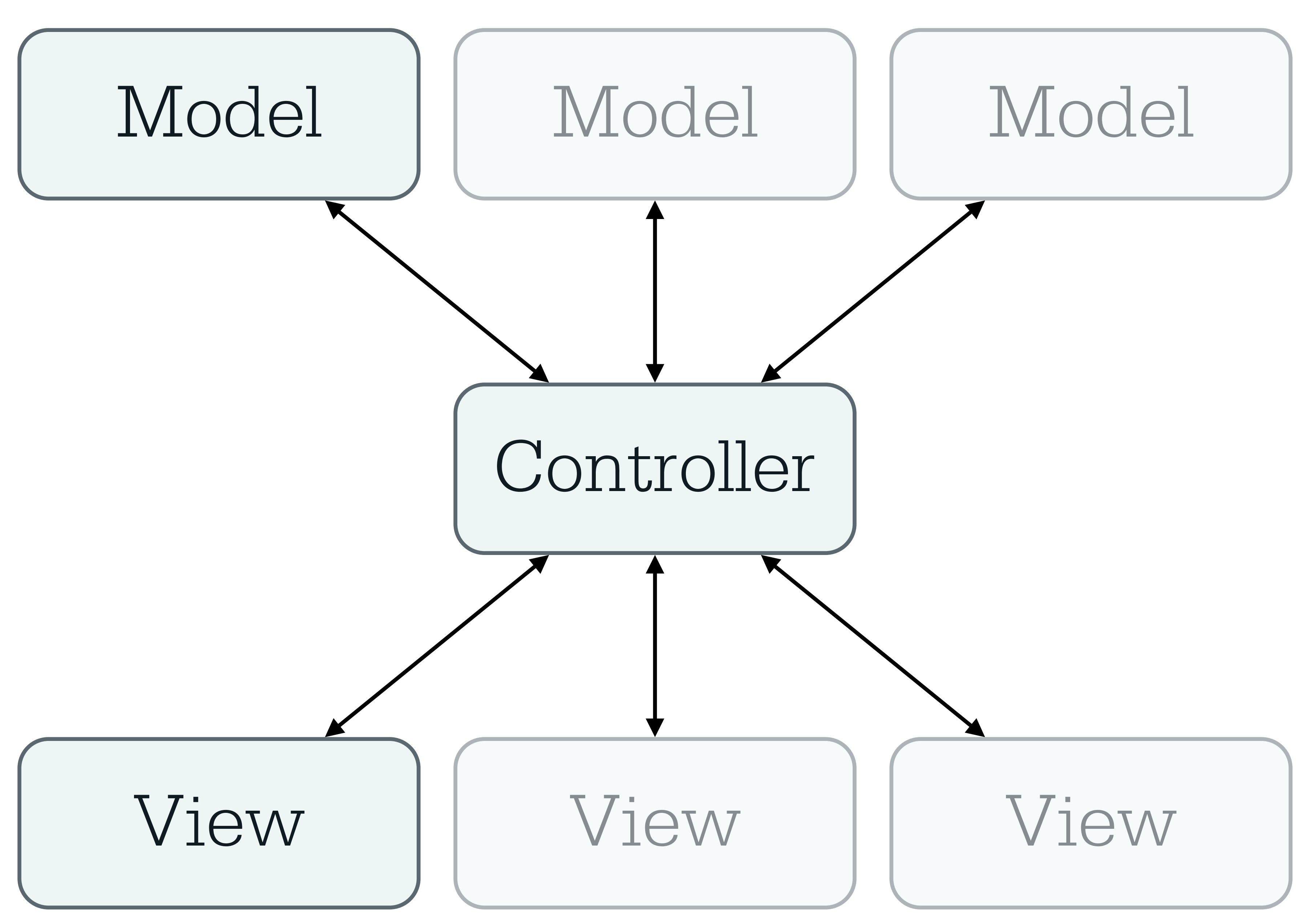
MVC STRUCTUR

Rant:

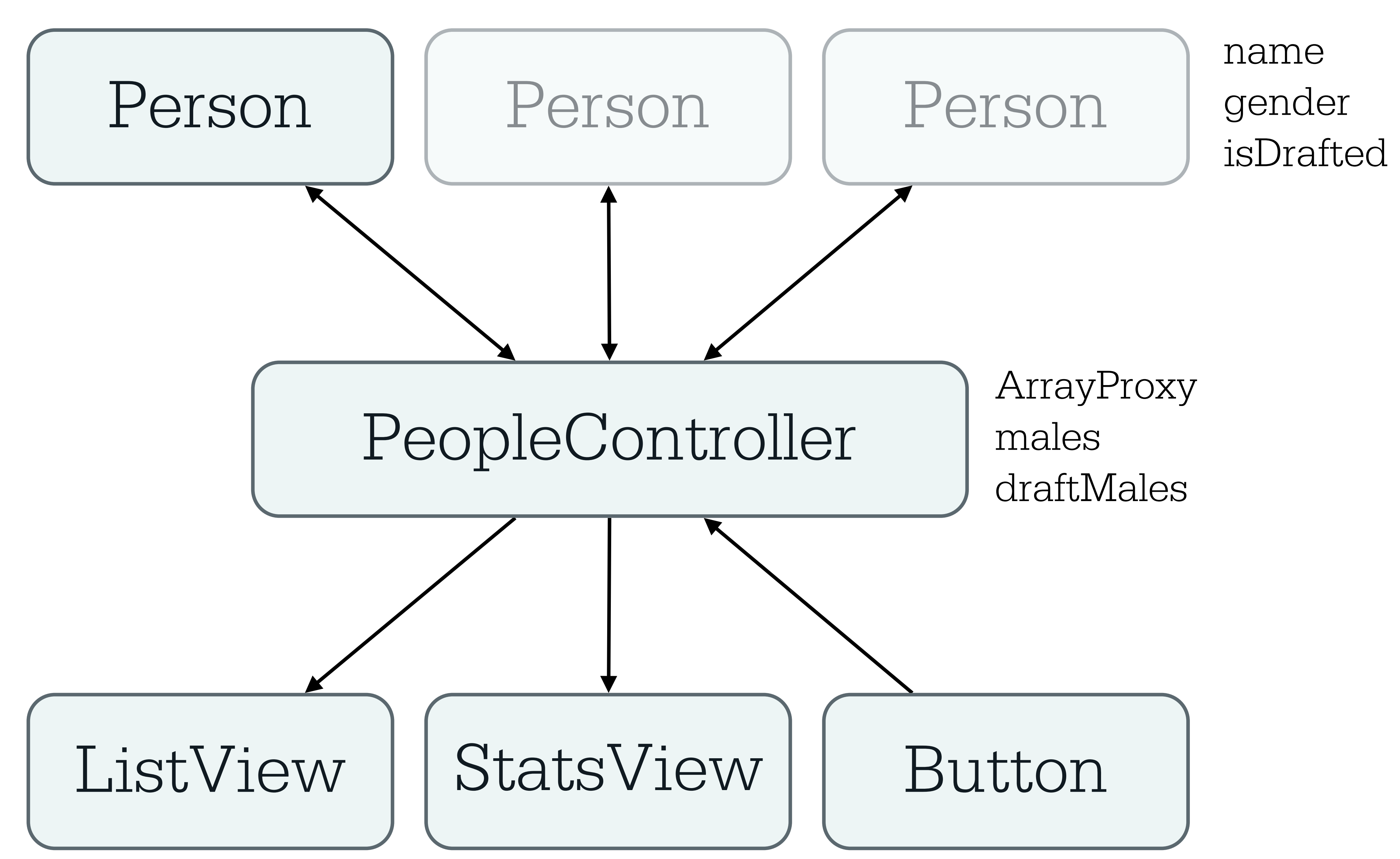
MVC/MVVM/
MVP/etc. all mean
the same thing

* Data
* View
* Intermediary

EMBER MVC.



ARRAY CONTROLLER.



OPTIONAL EMBER-DATA.

ROLLING IN EMERGING PATTERNS.

One of Ember's main goals to identify common patterns and make them conventional.

As patterns emerge, we work to reduce the boilerplate necessary to implement them.

WHY?

MANAGES COMPLEXITY.

Ember aims to reduce the complexity created by the links between many moving pieces by clearly defining how communication occurs between parts of your application.

MAKES YOU MORE PRODUCTIVE.

Ember's philosophy is that by eliminating trivial choices and making the answers conventional, you can focus your energy on non-trivial problems.

This philosophy is taken directly from Rails.

BUILT FOR THE LONG HAUL.

Ember is not a throwaway weekend project or a corporate-sponsored project.

It is built by and for the Ember community, an open source project first and only.

OPTIMIZED FOR DEVELOPER HAPPINESS.

OBJECT MODEL.

CLASSES.

```
Person = Ember.Object.extend({
  firstName: null,
  lastName: null
});
```

MIXINS.

```
Speaker = Ember.Mixin.create({
  hello: function() {
    var first = this.get('firstName'),
          last = this.get('lastName');

    alert(first + " " + last + ": HELLO!")
  }
});

Person = Ember.Object.extend(Speaker);

Person.create({
  firstName: "Yehuda",
  lastName: "Katz"
}).hello();
```

MIXINS.

```
Speaker = Ember.Mixin.create({
  hello: function() {
    var first = this.get('firstName'),
        last = this.get('lastName');

    alert(fi                         LO!")
  }
});

Person = Ember.Object.extend(Speaker);

Person.create({
  firstName: "Yehuda",
  lastName: "Katz"
}).hello();
```

Yehuda Katz: HELLO!

OK

MIXINS AND SUPER.

```
Speaker = Ember.Mixin.create({
  hello: function() {
    var first = this.get('firstName'),
        last = this.get('lastName');

    return first + " " + last + ": HELLO";
  }
});

Dog = Ember.Object.extend(Speaker, {
  hello: function() {
    return this._super() + " THIS IS DOG";
  }
});

var phil = Dog.create({
  firstName: "Budweiser",
  lastName: "Phil",

  hello: function() {
    return this._super() + " ZAAAAAAAA";
  }
});

alert(phil.hello());
```

MIXINS AND SUPER.

```
Speaker = Ember.Mixin.create({
  hello: function() {
    var first = this.get('firstName'),
        last = this.get('lastName');

    return first + " " + last + ": HELLO";
  }
});

Dog = Ember.Object.ex
  hello: function() {
    return this._supe
  }
});

var phil = Dog.create
  firstName: "Budweiser",
  lastName: "Phil",

  hello: function() {
    return this._super() + " ZAAAAAAAA";
  }
});

alert(phil.hello());
```

Budweiser Phil: HELLO THIS IS DOG ZAAAAAAAA

OK

COMPUTED PROPS.

```
Person = Ember.Object.extend({
  fullName: function() {
    return this.get('firstName') +
      ' ' + this.get('lastName');
  }.property('firstName', 'lastName')
});

var yehuda = Person.create({
  firstName: "Yehuda",
  lastName: "Katz"
});

alert(yehuda.get('fullName'));
```

COMPUTED PROPS.

```
Person = Ember.Object.extend({
  fullName: function() {
    return this.get('firstName') +
      ' ' + this.get('lastName');
  }.property('firstName', 'lastName')
});

var yehuda
  firstName: "Yehuda",
  lastName: "Katz"
});

alert(yehuda.get('fullName'));
```

Yehuda Katz

OK

UNIFORM ACCESS IS GREAT FOR REFACTORING.

IT ALSO GIVES BINDINGS A SINGLE RULE.

NAMESPACES.

```
>> Core = Ember.Namespace.create();
>> Core.Person = Ember.Person.extend();
>> Core.Person.toString();
=> Core.Person
>> Core.Person.create().toString();
=> <Core.Person:ember157>
```

MASSIVELY IMPROVED DEBUGGING.

CAN USE NAMES IN CONVENTIONS.

This is something Rails does a lot of, and which is hard to do in JS since klass.name is not available in general.

We use this extensively in the default REST adapter in ember-data.

CONVENTIONS.

```
App.Post = DS.Model.extend({
  title: DS.attr('string'),
  body: DS.attr('string')
});

// vs.

App.Post = DS.Model.extend({
  collectionURL: "/posts",
  singleURL: "/post",

  title: DS.attr('string'),
  body: DS.attr('string')
});
```

A SINGLE, RICH OBJECT MODEL.

THE BASIC RULES APPLY TO ALL KINDS OF OBJECTS.

LIKE BACKBONE EVENTS ON STEROIDS.

Our goal, again is to wrap up common patterns in the base distribution so that you don't have to think about them all the time.

THE VIEW LAYER.

DATA BOUND.

```
<ul>
  <li>First</li>
  <li>Second</li>
  <li>Third</li>
</ul>
```

DATA BOUND.

```
<ul>
  {{#each App.items}}
    <li>{{name}}</li>
  {{/each}}
</ul>
```

DATA BOUND.

```
App.items = [
  { name: "One" },
  { name: "Two" },
  { name: "Three" }
];
```

DATA BOUND.

```
App.items = [
  {
  {
  {
];
```

- One
- Two
- Three

DATA BOUND.

```
App.items.pushObject({
  name: "Fourth"
});
```

DATA BOUND.

```
App.items.pushObject({
  n
});
```

- One
- Two
- Three

DATA BOUND.

```
App.items.pushObject({
  n
});
```

- One
- Two
- Three
- Fourth

DATA BOUND.

```
App.items
  .objectAt(0)
  .set('name', '1');
```

DATA BOUND.

```
App.items
```

- One
- Two
- Three
- Fourth

DATA BOUND.

```
App.items
```

- 1
- Two
- Three
- Fourth

CONDITIONALS.

```
{{#if App.isLoaded}}
  <div class="content">
    <h1>{{App.title}}</h1>
    <div>{{App.body}}</div>
  </div>
{{else}}
  <img src="/images/loading.gif">
{{/if}}
```

ASYNCHRONY.

```
App = Ember.Application.create();

jQuery.getJSON("/bootstrap",
  function(json) {
    App.set('title', json.title);
    App.set('body', json.body);
    App.set('isLoaded', true);
  });
```

DECLARATIVE.

```
{{#if App.isLoaded}}
  <div class="content">
    <h1>{{App.title}}</h1>
    <div>{{App.body}}</div>
    {{#view App.DashboardView}}
      {{#each App.widgetsController}}
        {{view App.Widget widgetBinding="this"}}
      {{/each}}
      {{#view Ember.Button
              target="App.widgetsController"
              action="newWidget"}}
        New Widget!
      {{/view}}
    {{/view}}
  </div>
{{else}}
  <img src="/images/loading.gif">
{{/if}}
```

COMPOSABLE.

```
{{#if App.isLoaded}}
  <div class="content">
    <h1>{{App.title}}</h1>
    <div>{{App.body}}</div>
    {{view App.DashboardView templateName="dashboard"}}
  </div>
{{else}}
  <img src="/images/loading.gif">
{{/if}}

<!-- dashboard template -->
{{#each App.widgetsController}}
  {{view App.Widget widgetBinding="this"}}
{{/each}}
{{#view Ember.Button
        target="App.widgetsController"
        action="newWidget"}}
  New Widget!
{{/view}}
```

CUSTOMIZABLE.

```
App.ButtonView = Ember.View.extend({
  tagName: "button",
  didInsertElement: function() {
    this.$().button();
  }
});
```

CUSTOMIZABLE.

```
{{#view App.ButtonView}}Hi!{{/view}}
```

CUSTOMIZABLE.

```
App.ButtonView = Ember.ButtonView.extend({
  didInsertElement: function() {
    this.$().button();
  }
});
```

Ember's controls are UI-agnostic, so you can use the semantics (like target/action, text field) without having to back out Ember-supplied UI.

CUSTOMIZABLE.

```
{{#view App.ButtonView
        target="App.widgetController"
        action="new"}}
  New Widget
{{/view}}
```

TEMPLATES.

```
<body>
  <script type="text/x-handlebars">
    <!-- view contents here -->
  </script>
</body>
```

NAMED TEMPLATES.

```
<head>
  <script type="text/x-handlebars"
          data-template-name="name">
    <!-- view contents here -->
  </script>
</head>
```

TEMPLATES.

```
Ember.TEMPLATES["name"] =
  Ember.Handlebars.compile(string);
```

TEMPLATE API.

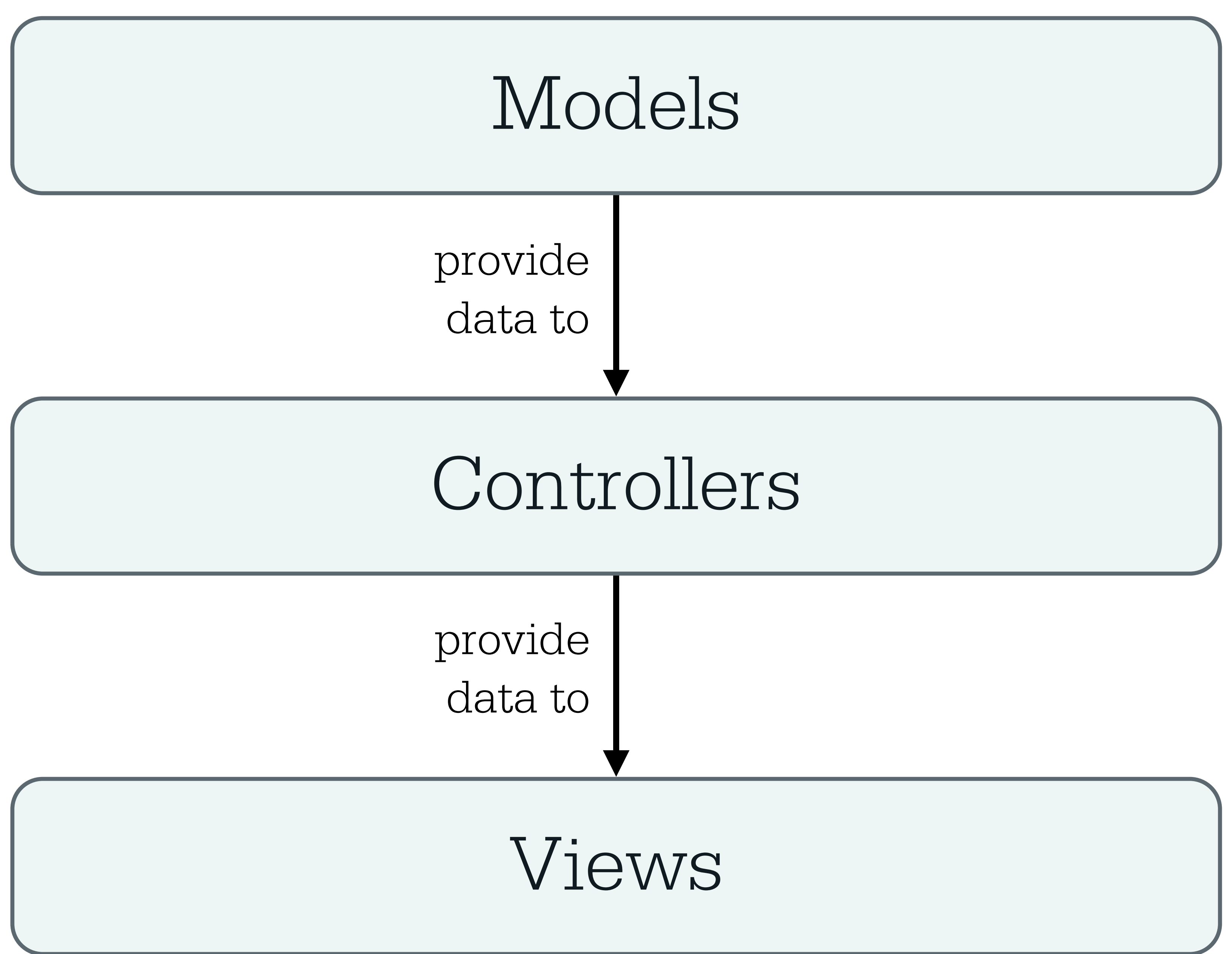
```
Ember.TEMPLATES["name"] =
  Ember.Handlebars.compile(string);

Ember.TEMPLATES["other"] =
  function() { return "hi!" };
```

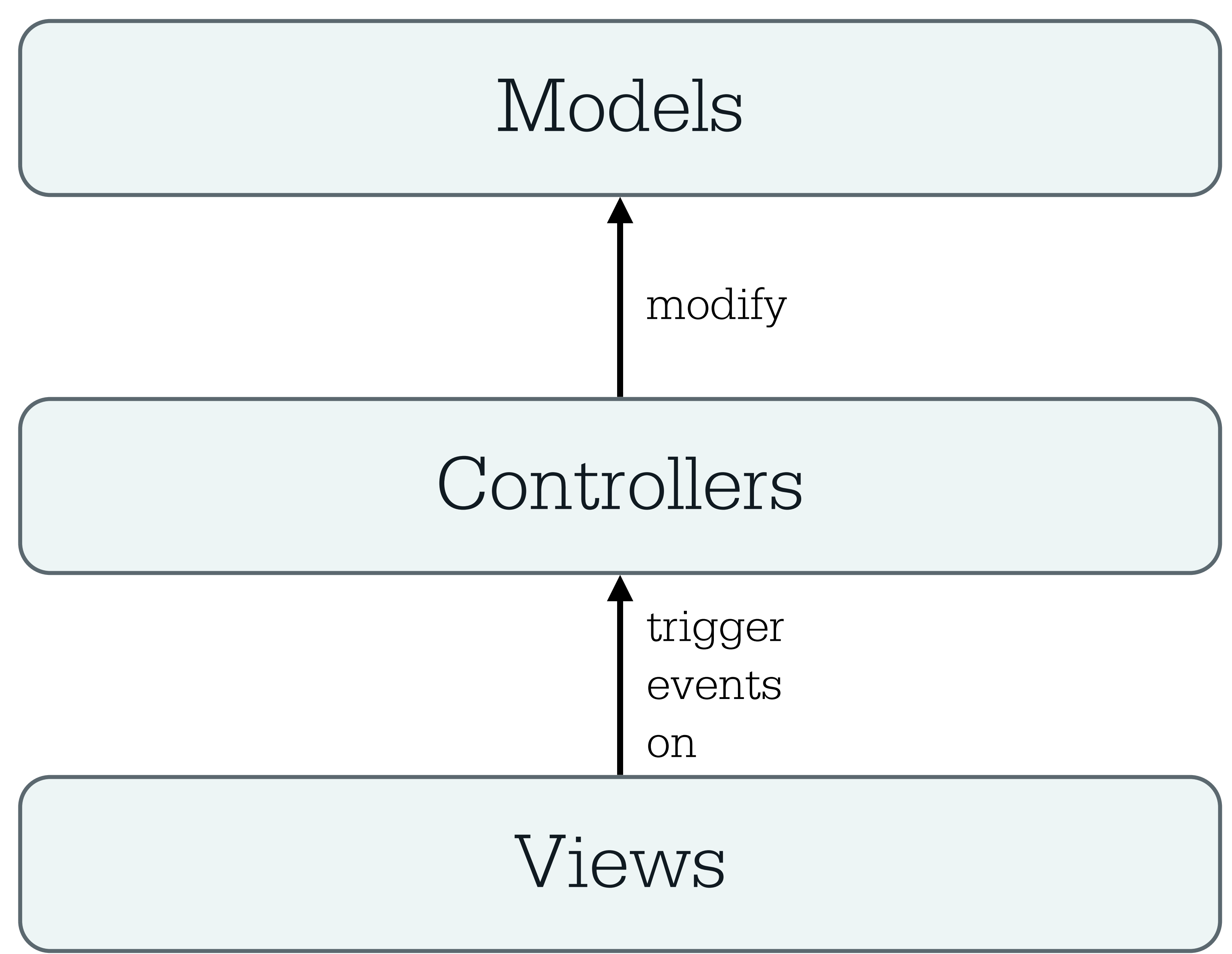
You will need to do property lookups through .get, so at least a bit of template engine customizability is required. If arbitrary code is allowed, you can even implement observable helpers.

BASIC APP ARCHITECTURE

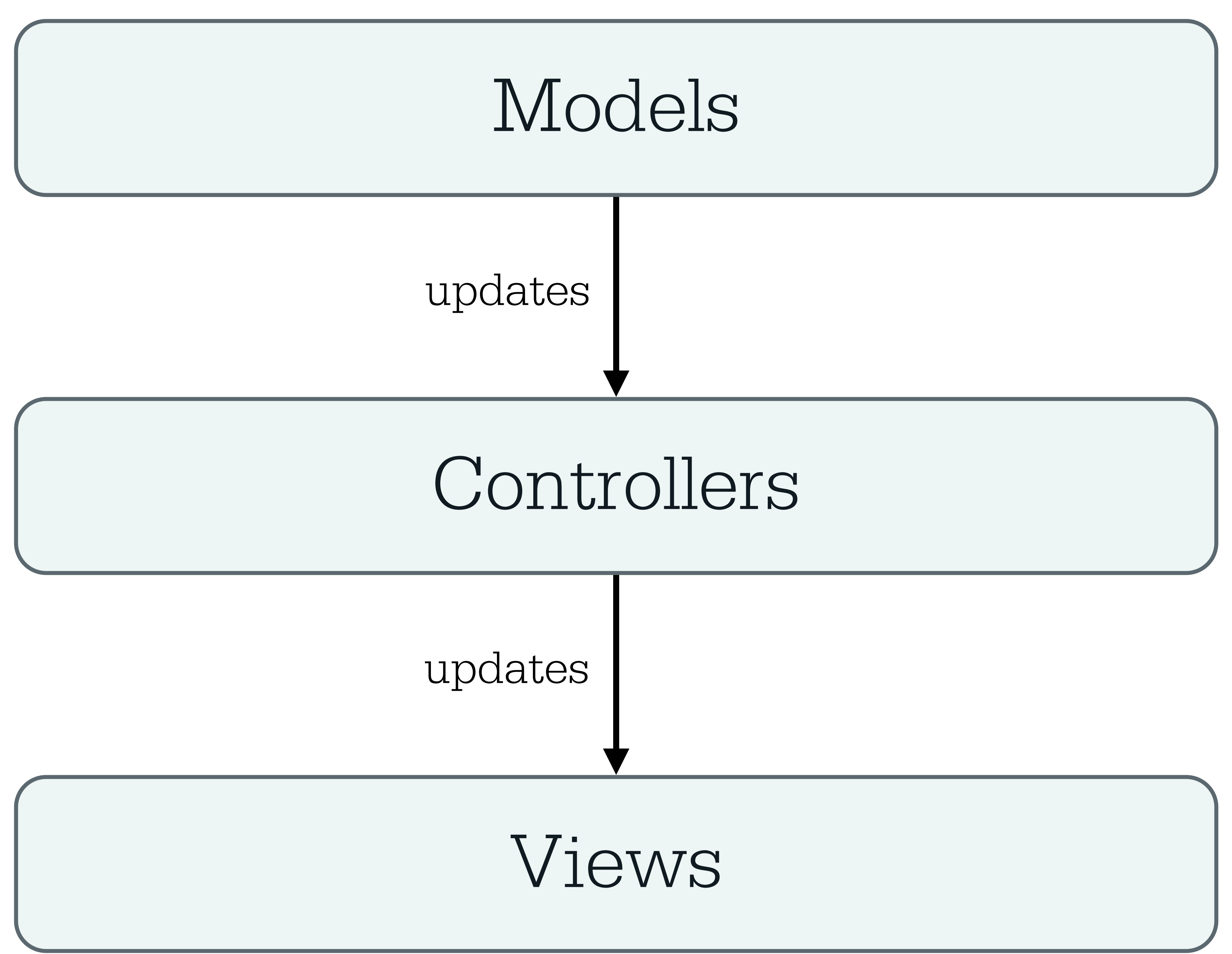
ARCHITECTURE.



ARCHITECTURE.

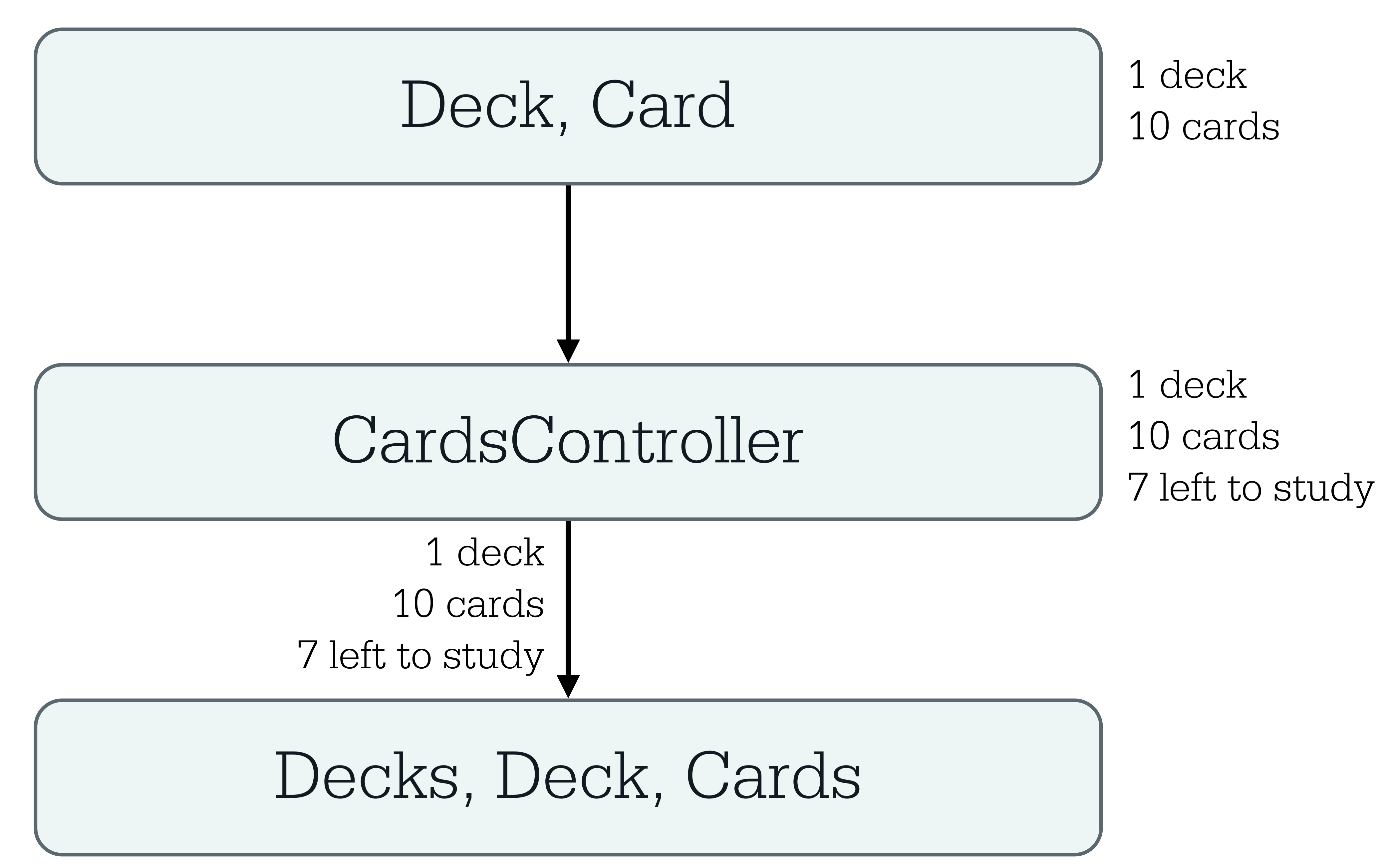


ARCHITECTURE.

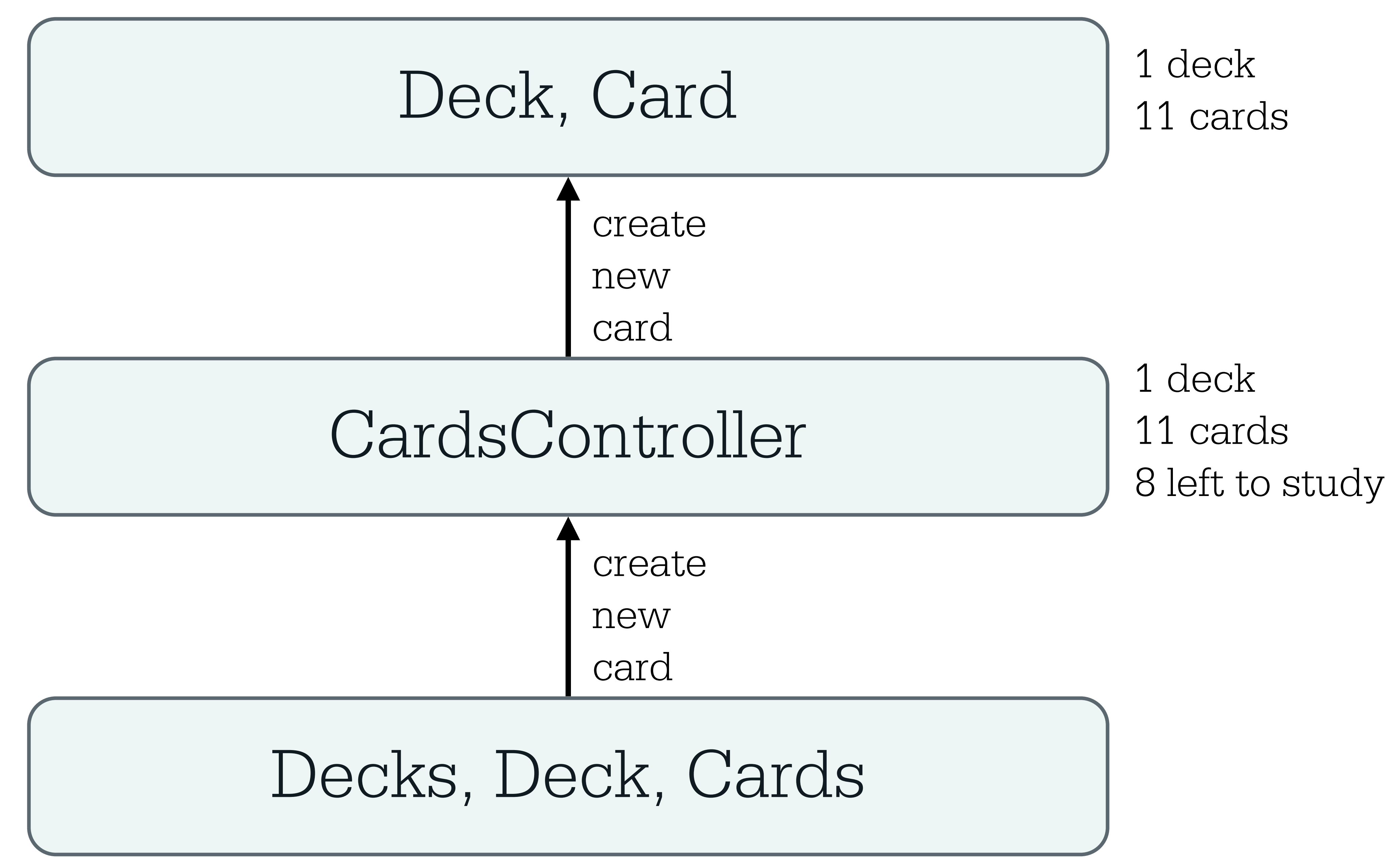


EXAMPLE.

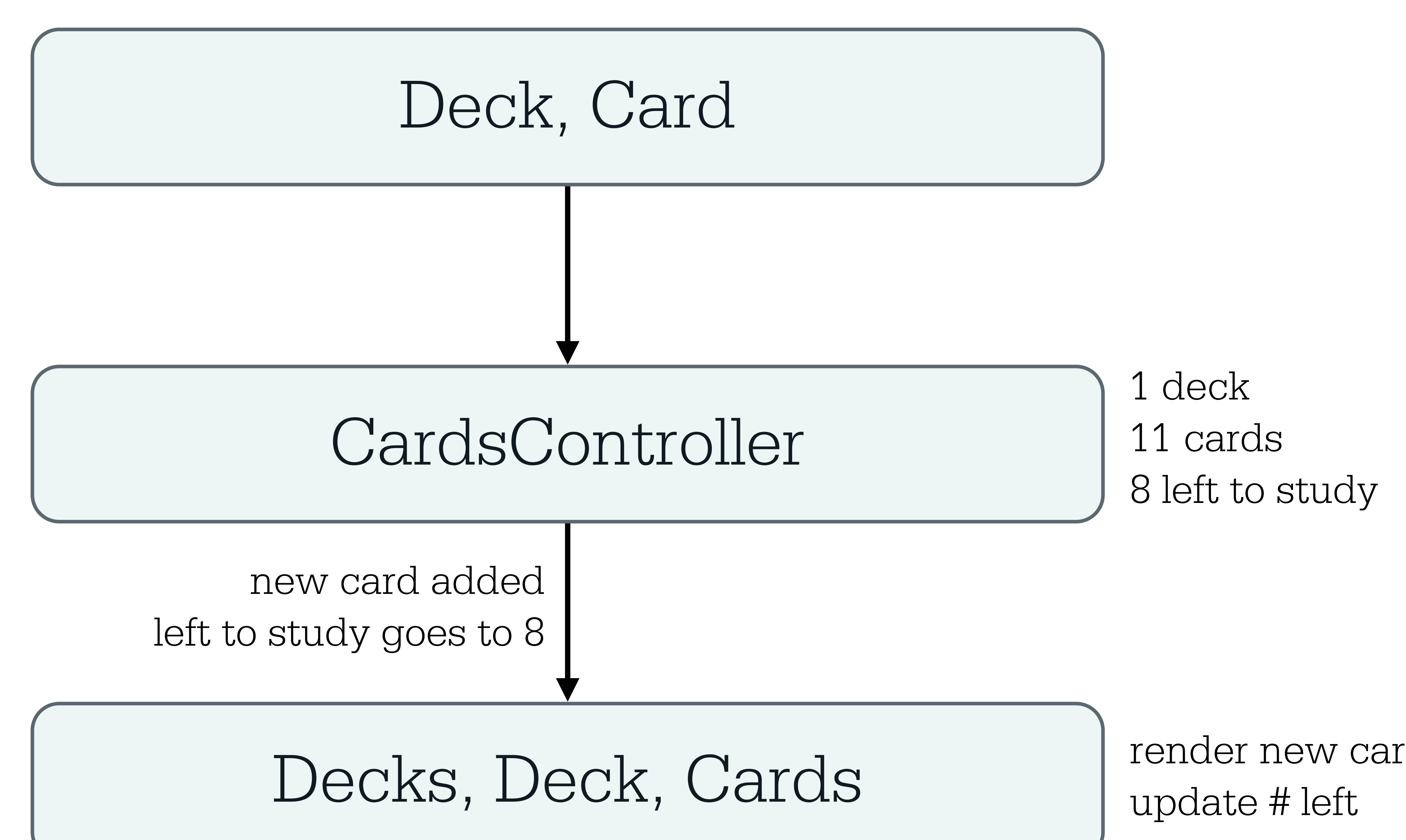
FLASHCARDS.



FLASHCARDS.



FLASHCARDS.



RULE OF THUMB.

RULE OF THUMB.

- Models contain persistable data
- Controllers proxy models and add client-side application state
- Views represent a single representation of persistable or application state

EVOLVING PATTERNS

ROUTING.

STATE MANAGERS.

PERSISTENCE.

IMPROVED DEBUGGING.

Deferred, asynchronous code can be hard to debug (what caused that callback to be triggered?!)

A debug build could provide more information, at the cost of slower performance. We're also working on and off on a Chrome extension.

DOCUMENTATION.

Our documentation is sparse and currently focused on the top-down. Generated documentation is coming, as is more information on common patterns not yet encapsulated in code.

RAILS INTEGRATION.

Ember works well with any (or no) server-side framework. Because Rails and Ember share philosophies, we can leverage conventions on both sides to further reduce boilerplate for people willing to adhere to even more strict rules about file structure and REST APIs.

AS PATTERNS SOLIDIFY, WE ROLL THEM IN.

SOMETIMES WE GIVE THEM A LITTLE PUSH.

THANK YOU.

@WYCATS

@EMBERJS
EMBERJS.COM
GITHUB.COM/EMBERJS
IRC #EMBERJS