MV*

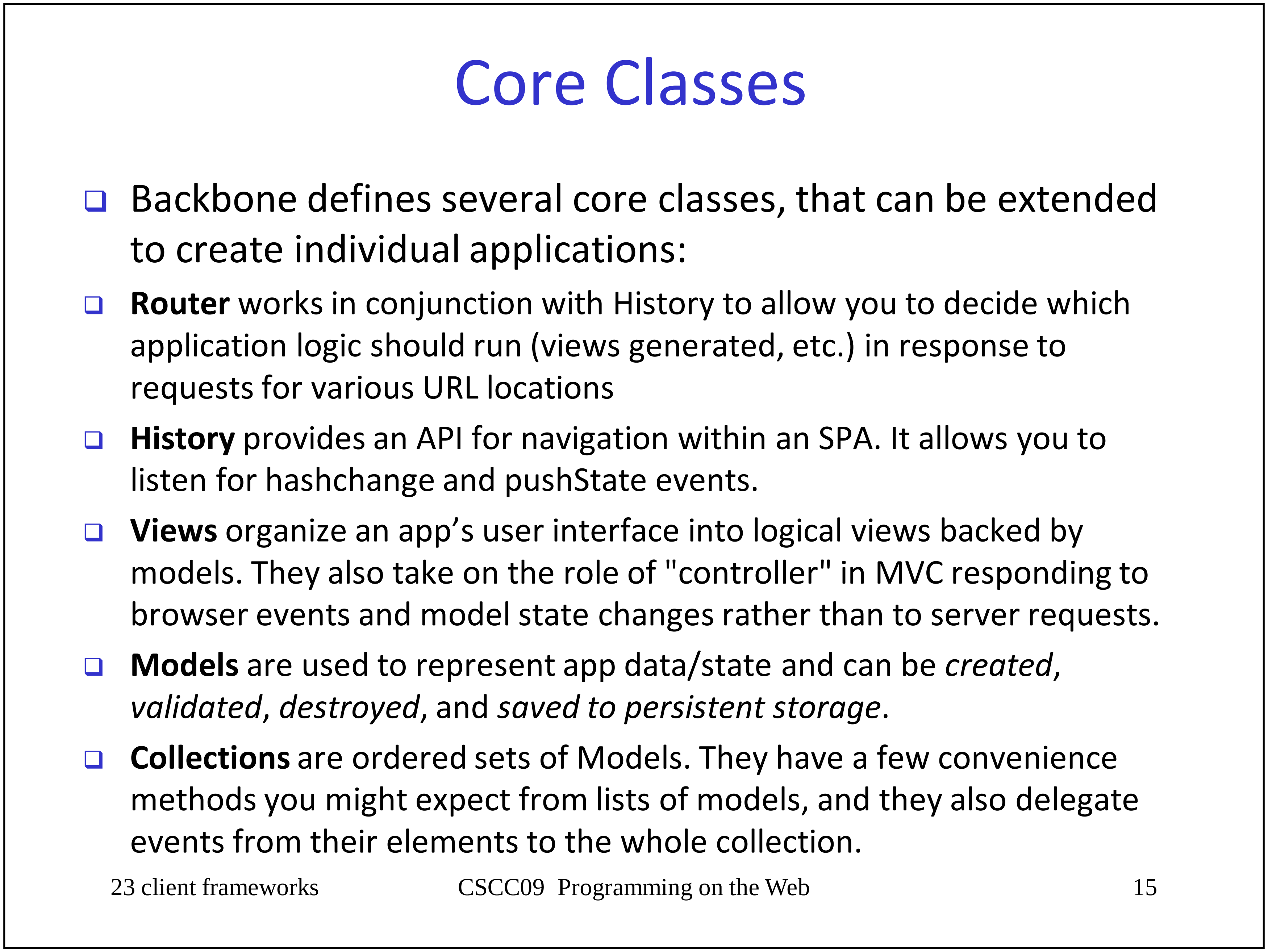
- Backbone is a client-side MV* “framework” or library.
  - We say MV* because Backbone does not provide an explicit controller component
  - Rather, controller responsibilities are handed by a combination of Router and View components
    - Routers have routing-handlers to respond to URL changes with view updates
    - Views have event-handlers to manage DOM interactions
  - Models and collections serve as client-side datastores
  - Views render templates and represent various parts of the app UI

MV* concepts

- Router (plays part of MVC Controller role)
  - Handles SPA navigation by responding to URL changes
- View
  - Packages presentation details into organized sub-views
- Model/Collection
  - Abstracts persistent remote/network resources (typically DB) for easy/fast local access
  - Syncronizing save() updates remote DB to match local model
- Event binding
  - model/collection state changes trigger events

Dependencies

- What does Backbone depend on?
  - Underscore.js is the only hard dependency
  - But must also supply a DOM-manipulation library, like jQuery
- Can I still use jQuery?
  - Yes, jQuery and Backbone are compatible and to some extent complementary
  - You can continue to use jQuery for tasks where it excels, such as referencing or manipulating parts of the DOM
  - Use Backbone to define the overall structure of your JavaScript code (Router, Views, Models, etc) and utilizing its built-in tools for common tasks such as data sync'ing, routing, event-propagation, etc.



Core Classes

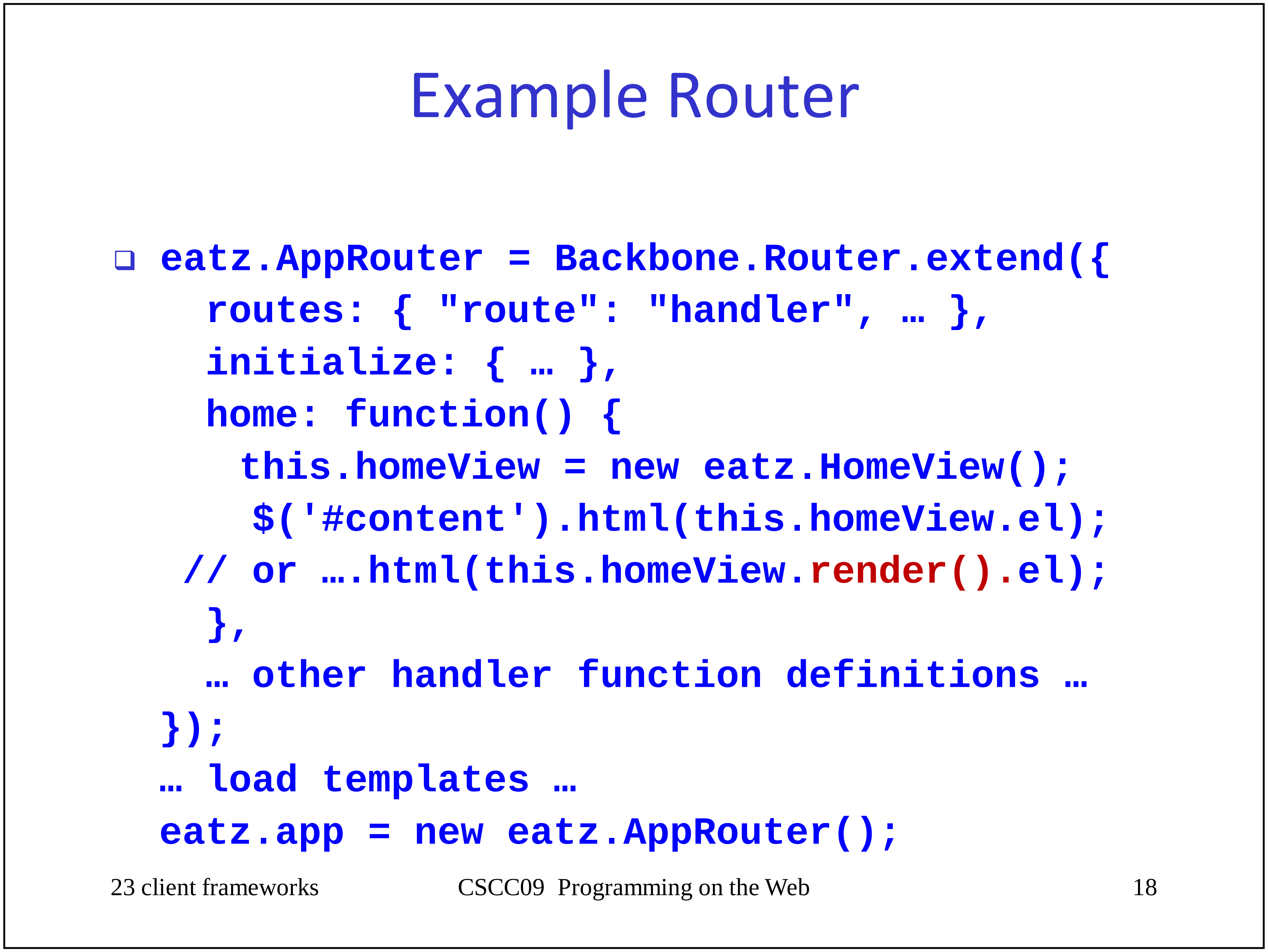
- Backbone defines several core classes, that can be extended to create individual applications:
- **Router** works in conjunction with History to allow you to decide which application logic should run (views generated, etc.) in response to requests for various URL locations
- **History** provides an API for navigation within an SPA. It allows you to listen for hashchange and pushState events.
- **Views** organize an app’s user interface into logical views backed by models. They also take on the role of "controller" in MVC responding to browser events and model state changes rather than to server requests.
- **Models** are used to represent app data/state and can be *created*, *validated*, *destroyed*, and *saved to persistent storage*.
- **Collections** are ordered sets of Models. They have a few convenience methods you might expect from lists of models, and they also delegate events from their elements to the whole collection.

Routers

- Associate URL-fragments with app functionality
- They key on hashchange events (content after # in URL) to trigger a route-dispatch
  - Whenever the browser address-bar URL changes, the router will be invoked
  - To support SPA-type behavior, the app should change only the hash portion of the URL, otherwise a server-request is triggered
- Hashstate URL's are stateful and bookmarkable

Routers

- HTML5 **`pushState`** supports app views without the use of a hash in the URL
  - depends on a server that is able to respond to these view-oriented requests
- Perform page routing for a SPA – i.e. binding URL-changes to page-view changes
- Implement control flow – code can explicitly change the URL, and thus transition to a different view (different part of app functionality)
- Don’t confuse client-side routing in Backbone with server-side routing



Example Router

```
❑ eatz.AppRouter = Backbone.Router.extend({
    routes: { "route": "handler", … },
    initialize: { … },
    home: function() {
      this.homeView = new eatz.HomeView();
       $('#content').html(this.homeView.el);
   // or ….html(this.homeView.render().el);
    },
    … other handler function definitions …
  });
  … load templates …
  eatz.app = new eatz.AppRouter();
```

23 client frameworks CSCC09 Programming on the Web 18

Views

- Although it might seem logical that a View would consist of HTML markup, in practice the HTML markup for views is usually contained in separate <u>template</u> files, while the View itself contains the <u>logic</u> associated with the view:
  - how to initialize the view
  - how to render the view (the DOM-elements comprising the view) from template and model data
  - identifying the DOM element that will contain the view’s markup
  - DOM and model-change events the view should respond to, and the details of these responses
  - actions to take when the view is removed

View Example

```
eatz.HomeView = Backbone.View.extend({

    initialize: function () {
        this.render();  // omit if caller renders
    },

    render: function () {
        // create DOM content for HomeView
        this.$el.html(this.template());
        return this;    // support chaining
    }
});
```

Views

- When a view is created:
  - DOM element (el) is created, if not supplied as a parameter
  - **`initialize()`** method is invoked
  - “events” object is parsed and the listed methods are registered as event-handlers
- Once the view exists, its caller (or possibly the view itself):
  - invokes **`render()`**, which looks up its template, generates an HTML string from a JSON representation of its model, and returns **`this`** (the view), to support method chaining
  - the value returned by **`render()`** is inserted in the DOM, typically to an existing element visible in the browser, which will in turn cause the View to become visible in the browser

Memory Leaks

- When a view is replaced by the router, that view still exists in the DOM (recall that a view is just a DOM structure), as do any events bound to that view
- Consequences?
  - DOM memory gradually accumulates “ghost” views
  - Event listeners still reference these “ghost” views
- The underlying problem is that JavaScript’s built-in garbage-collection mechanism won’t remove DOM nodes so long as references to them still exist in scope

Memory Leaks

- How to avoid this problem?
  - Remove view from DOM when switching to new view: `Backbone.View.remove()`
  - Unbind events bound to a view: `Backbone.View.unbind()`
- The above practices are a good start, but may not always be sufficient, e.g.
  - If a model is associated with the view, and has an event such as a change registered to trigger re-rendering, must unbind this event
  - `this.model.unbind('change', this.render, this);`
  - If jQuery holds a reference to `this.$el`, have to delete that

Templates

- Backbone could generate all necessary HTML markup dynamically; however, such tight coupling of code and page-markup is undesirable for several reasons:
  - not very readable: you have to mentally parse out the JavaScript from the HTML
  - cumbersome to maintain: if your Web designers want you to change the look of an app, you have change code
  - not well suited to partitioning of effort: you want to divide up the work of designing app-layout (look) from behavior (logic) among different people or teams, but having both in the same code makes this problematic
- Templates solve the above problems by injecting model values into HTML markup skeletons at designated points

Templates

- Underscore's _.template function has this form:

  ```
  _.template(templateString, data?, settings?)
  ```

  Where **`templateString`** is an HTML template, optional **`data`** object specifies field values, and optional **`settings`** override global settings, for example:

  ```
  _.template("Hello <%=user%>!", { user:"Joe" })
  ```

  results in value: `'Hello Joe!'`

  If **`data`** is omitted, the result of `_.template(...)` is a <u>function</u>, as in:

  ```
  var tpl = _.template("Hello <%= user %>!");
  tpl({user: "Joe"});  // bind data to variables
  ```

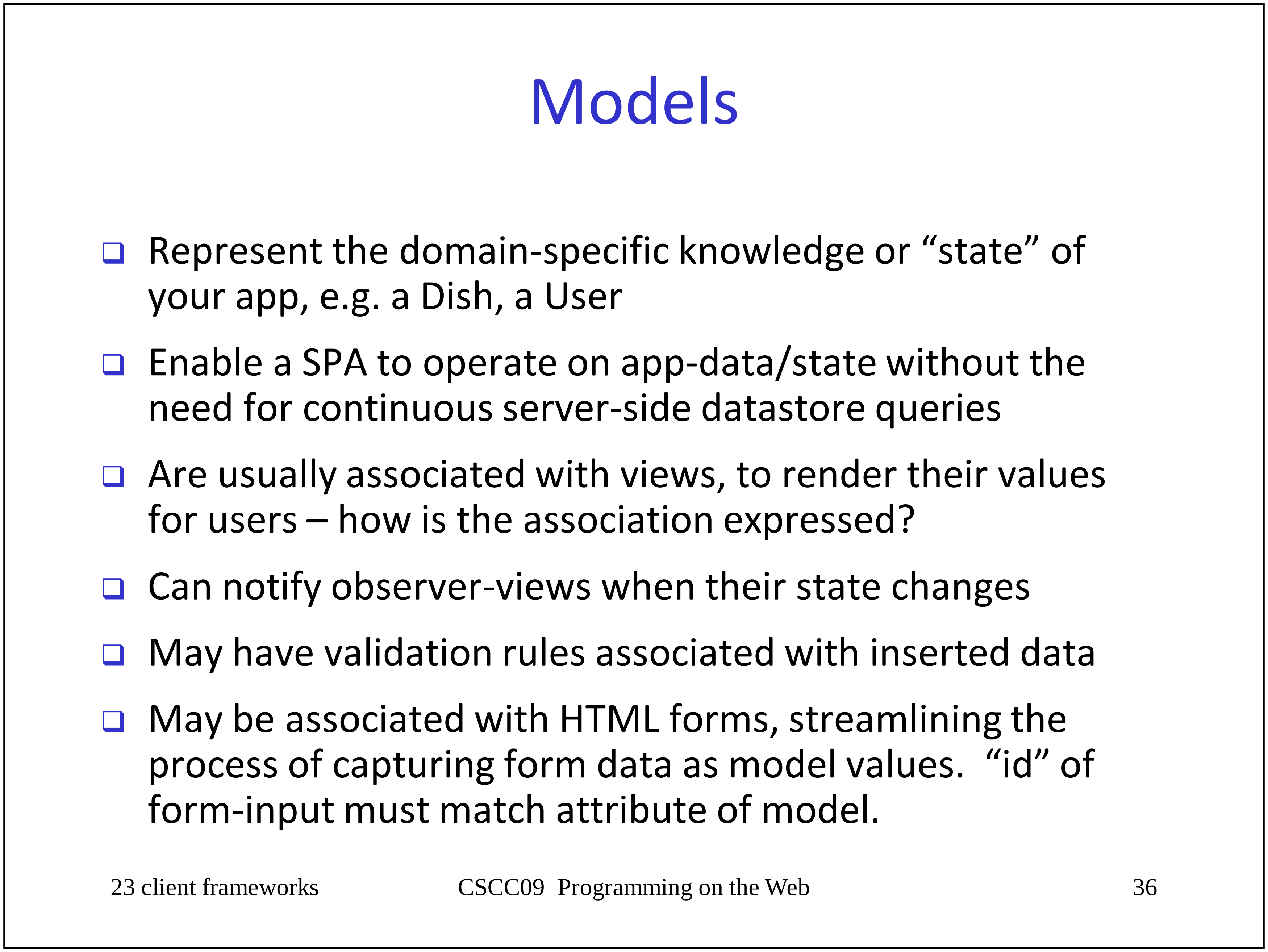
- Which results in HTML value: `'Hello Joe!'`

Template Example

- Example template from Assignment: EditView.html fragment:

```
 ...
   <div class="control-group">
     <label for="name" class="control-label"></label>
        <div class="controls">
           <input type="text" value="<%= street %>"/>
           <span class="help-inline"> </span>
        </div>
   </div>
...
```

model attribute



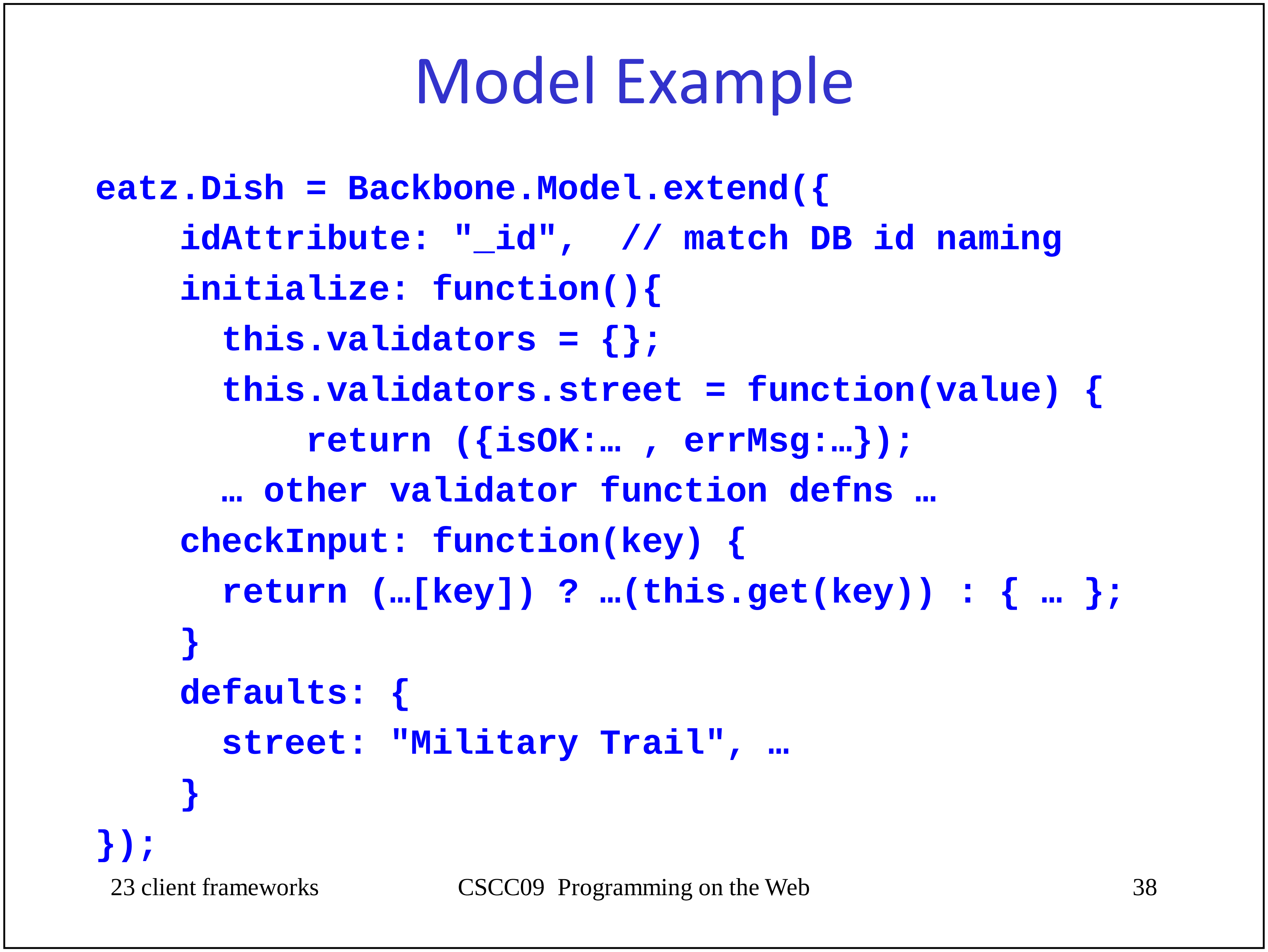
Models

- Represent the domain-specific knowledge or “state” of your app, e.g. a Dish, a User
- Enable a SPA to operate on app-data/state without the need for continuous server-side datastore queries
- Are usually associated with views, to render their values for users – how is the association expressed?
- Can notify observer-views when their state changes
- May have validation rules associated with inserted data
- May be associated with HTML forms, streamlining the process of capturing form data as model values. “id” of form-input must match attribute of model.

23 client frameworks CSCC09 Programming on the Web 36

Models

- Define attributes that extend Backbone.Model, similar to the idea of class-inheritance, but in this case we are adding attributes to a cloned Backbone.Model object
- Model objects can be grouped together into Collections, and can be persisted to a client or server-side datastore
- RESTful API's used to persist models across app uses
  - **`save()/fetch()`** to a RESTful-API datastore to persist/retrieve model values



Model Example

```
eatz.Dish = Backbone.Model.extend({
    idAttribute: "_id",  // match DB id naming
    initialize: function(){
      this.validators = {};
      this.validators.street = function(value) {
          return ({isOK:… , errMsg:…});
      … other validator function defns …
    checkInput: function(key) {
      return (…[key]) ? …(this.get(key)) : { … };
    }
    defaults: {
      street: "Military Trail", …
    }
});
```

23 client frameworks CSCC09 Programming on the Web 38

Collections

- Ordered set of models
- Methods to manage models in collections and collection persistence

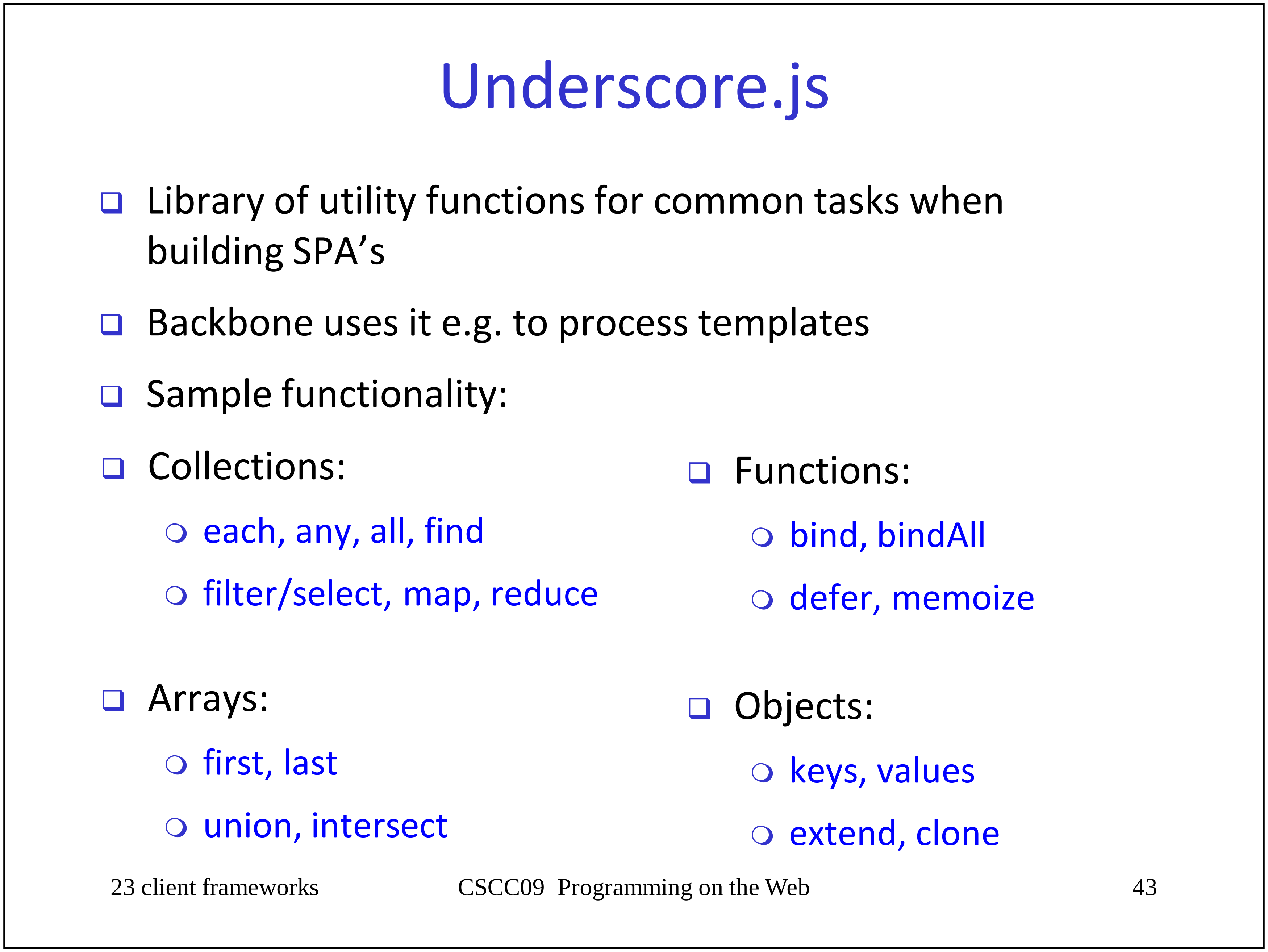
```
collection.add(models)  // add model to collection
collection.set(models)  // set model(s) attributes
collection.get(id)  // get model from collection
collection.fetch()  // update collection from server
collection.create(model) // add and persist model
```

Collections Example

```
eatz.Dishes = Backbone.Collection.extend({
    model: eatz.Dish,  // need to know model type

    // persist models in localStorage
    localStorage: new Backbone.LocalStorage('eatz')

    // alternatively, give server-API for persisting
    url: '...'
});
```



Underscore.js

- Library of utility functions for common tasks when building SPA’s
- Backbone uses it e.g. to process templates
- Sample functionality:
- Collections:
  - each, any, all, find
  - filter/select, map, reduce
- Arrays:
  - first, last
  - union, intersect
- Functions:
  - bind, bindAll
  - defer, memoize
- Objects:
  - keys, values
  - extend, clone