Tiger AppStore

WebApp

Steve Kehoe

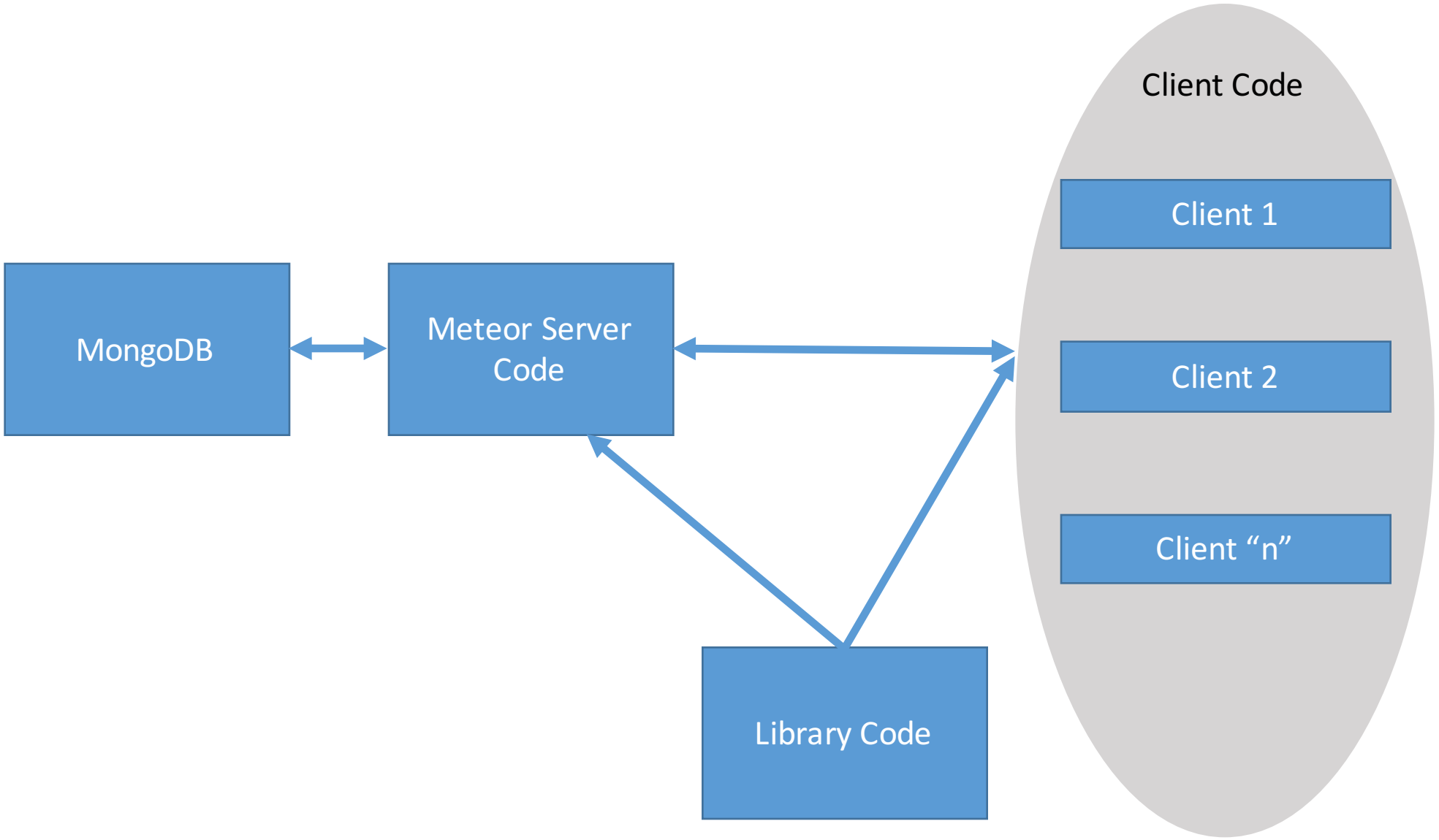
Goals For Today

1. Get an introduction to MeteorJS
2. Set up a dev environment for building web-apps
3. Learn how data is passed from sever to client
4. Learn how we can create dynamic html pages to display our data.

What is Meteor?

- Meteor is a web app framework built on top of NodeJS. This means we will use Javascript for both the client and server side code.
- Meteor uses Distributed Data Protocol to automatically synchronize data between the client and server.
- Meteor is ideal for rapid prototyping as the framework handles many routine tasks (like handling the database interactions and pushing data updates).
- Every time you save a file, Meteor automatically recompiles and uses “hot code push” to push the changes to all active clients!

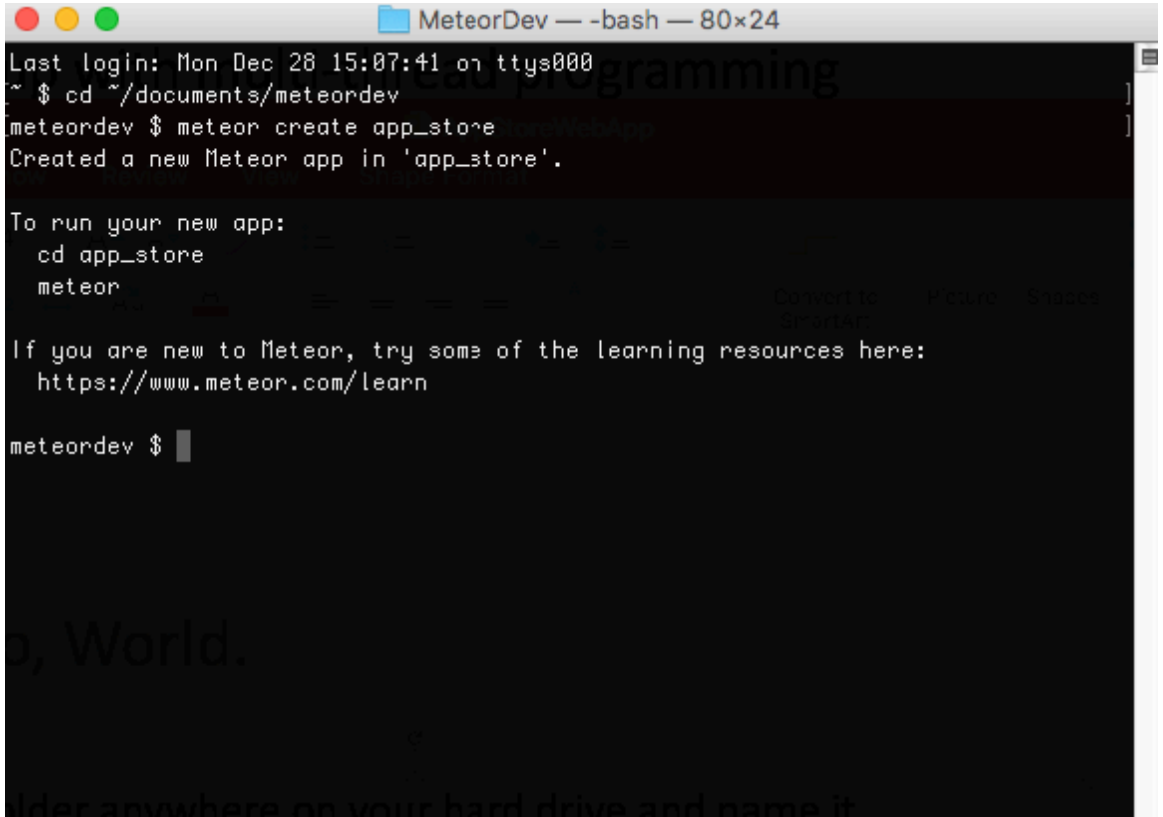
Meteor App Basic Architecture



WebApp
Meteor Intro

Exploring Meteor

1. Create a folder anywhere on your hard drive and name it “MeteorDev”. On my Macbook, I’ve created this folder under ~/documents.
2. Open a terminal (or on Windows, a command prompt) and use ”cd” to change to your new directory. Then enter the command “meteor create app_store”.

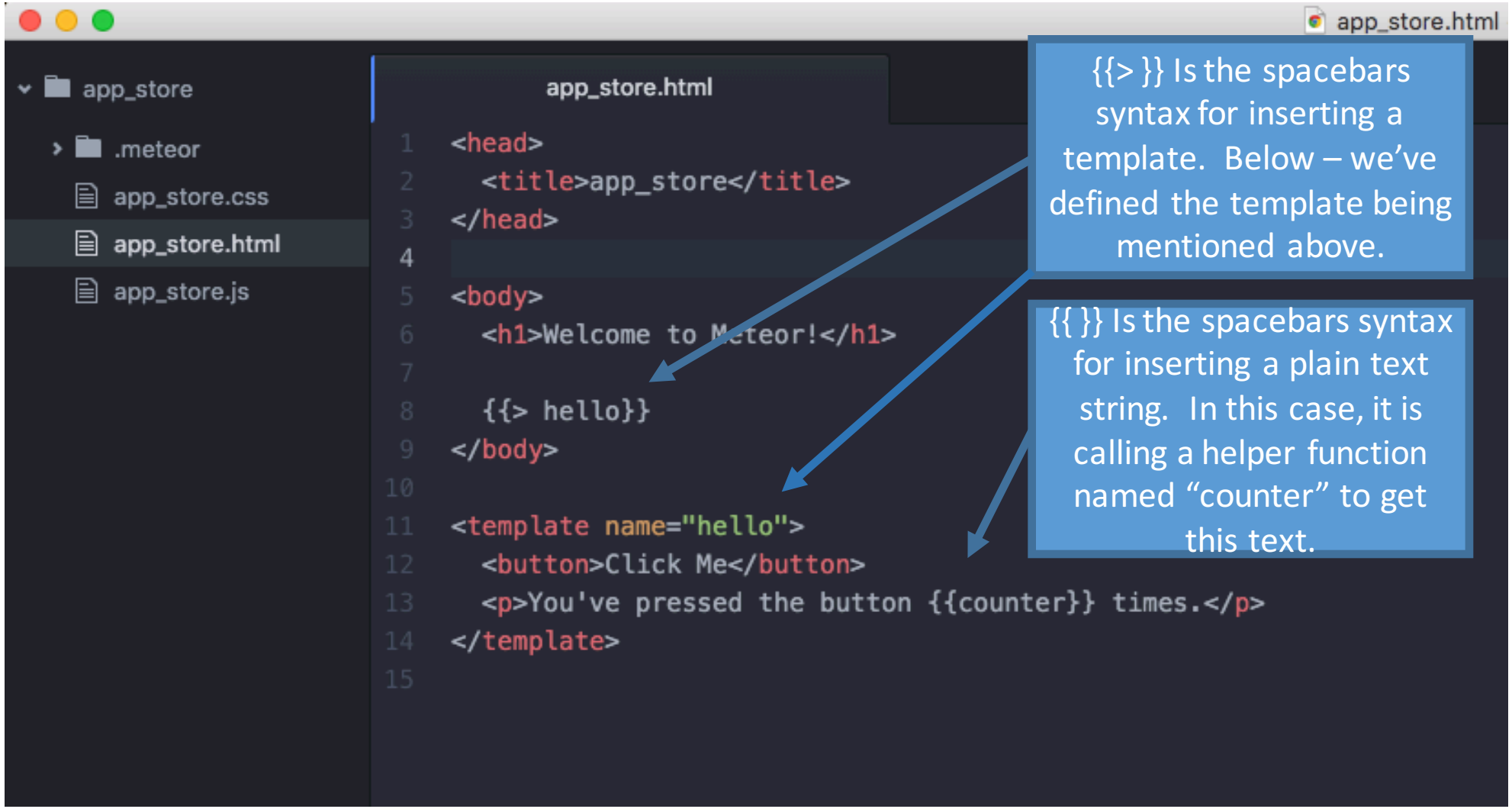


Exploring Meteor continued...

3. You can now type “cd app_store” to open the new folder that meteor created for you. By typing the “ls” command (or by opening the folder on your desktop) you will see that meteor has created three files for us: app_store.css, app_store.js, app_store.html. Type ”meteor run” in the console, and your new meteor app will be start up at http://localhost:3000. Navigate to this URL in your browser to see the starting state of the Meteor App.
4. Open app_store.html using your text-editor of choice.

Blaze Templates

This should mostly be a familiar HTML document – with the addition of Blaze and Spacebars
You can learn more about the Spacebars (syntax) side on their GitHub page.



Hello, World continued...

Now let's take a look at app_store.js

app_store
- .meteor
- app_store.css
- app_store.html
- app_store.js

app_store.html | app_store.css | app_store.js

```
1  if (Meteor.isClient) {
2    // counter starts at 0
3    Session.setDefault('counter', 0);
4
5    Template.hello.helpers({
6      counter: function () {
7        return Session.get('counter');
8      }
9    });
10
11   Template.hello.events({
12     'click button': function () {
13       // increment the counter when button is clicked
14       Session.set('counter', Session.get('counter') + 1);
15     }
16   });
17 }
18
19 if (Meteor.isServer) {
20   Meteor.startup(function () {
21     // code to run on server at startup
22   });
23 }
24
```

By wrapping this code in a Meteor.isClient if block, we ensure this code only executes on the client side.

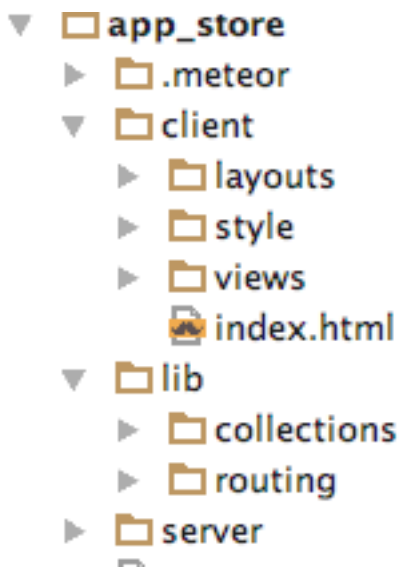
This helper function named "counter" is attached to the "hello" template. This is the function we are calling from app_store.html using {{counter}}

Meteor makes it even easier to attach events to DOM elements by allowing you to attach an events object to any blaze template. Everytime the button is clicked, a Session variable is updated.

Building the WebApp Server Side

Update Project structure

- By following Meteor's folder conventions we can control whether our code is available to the client, server, or both.
- Delete all starting files and create the directory structure pictured below.



Adding Packages

We will be using several third party packages to build our app store. Open a terminal and navigate to your app_store project directory.

Enter command:

'meteor add iron:router twbs:bootstrap barbatus:stars-rating'

And then:

'meteor remove autopublish insecure'

Here we've added three packages:

1. twbs:bootstrap - Twitter Bootstrap packaged for Meteor
2. iron:router – A Meteor package that handles routing between pages
3. barbatus:stars-rating – A small library to give us nice rating stars for the app store.

And the two default packages we removed:

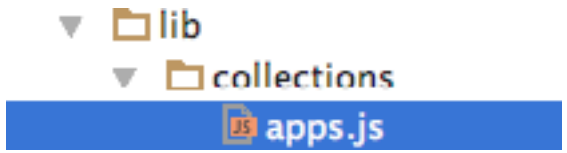
1. autopublish – a development package that publishes all of our MongoDB data to the client. Great for prototyping, insecure for production!
2. insecure – The package name says it all. This package allows the user client to create, update, read and delete any data in our database. It's another package meant to make development easier, but we won't have a need for this in our project.

Quick Collections Intro

- Collections are at the heart of MeteorJS and one of the reasons we love it as a quick prototyping framework. Meteor Collections are an extension of MongoDB Collections.
- Anything succesfully inserted into a collection on the server side is persisted in MongoDB automatically.
- Any collection (or part of a collection) that we publish to the client will automatically be cached on relevant pages that subscribe to it.
- All collections are reactively updated on both the client and the server. If a new app were published to our store, and thousands of clients were connected, all of their app stores would update immediately without any extra work on our end!

Creating the App Collection

We want our collections to be available on both the client and the server, so under the ‘lib’ directory we’ve also created a collections directory. Create a new file named ’app.js’ under the collections directory.



This file will only contain the following line:

```
Apps = new Meteor.Collection('apps');
```

Populating the App Collection

- Copy the data.json file your were given into your app's server folder.
- Copy the file data.json that is in the server folder into your apps server folder.
- We will now add another file to our server folder called fixtures.js and add the following code:

```
if(Apps.find({}).count() < 1){

   var fs = Npm.require('fs');

   fs.readFile('../../../../../server/data.json', 'utf8', Meteor.bindEnvironment(function(err,  data) {
      if (err) throw err;
      var newAppData = data.split("\n");

      for (var i = 0; i < newAppData.length - 1; i++) {
         var rawAppData = JSON.parse(newAppData[i]);
         var newApp = {};

         newApp.name = rawAppData.title;
         newApp.app_id = rawAppData.app_id;
         newApp.developer = rawAppData.developer;
         newApp.description  = rawAppData.intro;
         newApp.avgRating = parseInt(rawAppData.score) / 2;
         newApp.iconUrl  = rawAppData.thumbnail_url;
         newApp.reccomendedApps  = rawAppData.top_5_app;
         Apps.insert(newApp);
      }

   }, function(err){
      throw err;
   }));
}
```

Checks if our app collection is empty so we don't call this code on every run.

Reading in our json file using the Npm filesystem package.

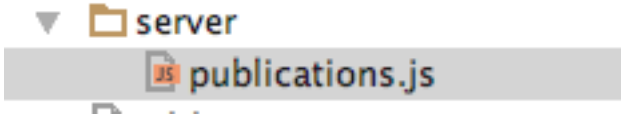
For each line in our data file, we will convert the string into a JSON object and map the object fields into the fields for our App Collection

Inserting the new apps into our collection

Publishing the App Collection

Now that we've loaded our data in from parts 1 and 2, we're ready to publish the App collection so that it will be available to the client side.

Under the 'server' folder create a file named publications.js



Add the following code:

```
Meteor.publish("apps", function (options) {
    return Apps.find({}, options);
});

Meteor.publish("singleApp", function (id) {
    return Apps.find({_id:id});
});

Meteor.publish("singleAppByAppId", function (appId) {
    return Apps.find({app_id:appId});
});
```

"apps" will return all Apps in the collection. It takes an options object that we'll use to push sorting and filtering operations onto the server side. This publication will be used in our Top Charts list.

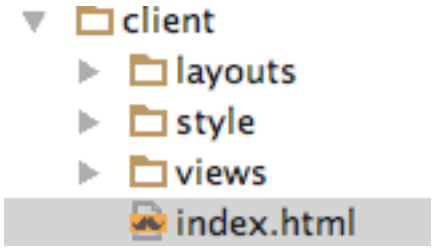
"singleApp" takes an appId as a parameter and returns just the one app that matches the appId. This publication will be used by our app details page for a single app.

"singleAppByAppId" is similar to the singleApp subscription except it takes the app_id that we got from the app store crawler in module #1. This will be used to look up the reccomended apps.

Building the WebApp Client Side

Client Side

Now we're ready to start working on the client side of our application. We can start by adding an index.html file in the client folder.



Add the following Code:

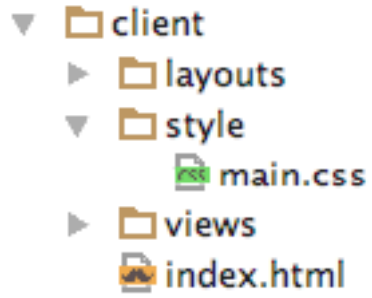
```
<head>
   <title>App Store</title>
</head>

<body>
</body>
```

We added this in order to add our App Title in the head section. Notice we have no <html> tags and we've left the body empty. Meteor's Blaze template engine will populate the body section, add additional links to the <head> section and wrap the entire document in HTML tags.

If you wanted to add any custom <head> content, such as favicons or touch screen icons, you would also do that here.

Add CSS Styles

To simplify the project, we've already provided some custom css for the app store in a file called main.css.

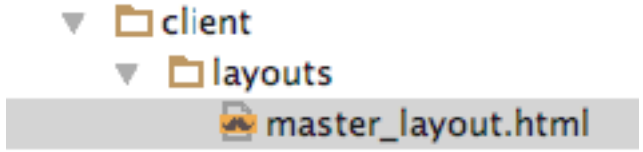


It's too long to paste here, so please copy the file from the cloned git repo over to your project.

Add Layout Template

One of the nice features of iron:router is the ability to have each of your view templates rendered inside a layout template.

Under client > layouts add a file master_layout.html with the following code.

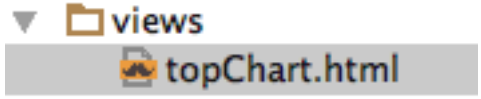



```
<template name="masterLayout">
   <div class="container">
      <div class="row">
         <div class="col-md-4 col-md-offset-4 col-sm-12 col-xs-12 mainContainer">
            {{> yield }}
         </div>
      </div>
   </div>
</template>
```

The {{> yield}} command tells iron:router where to render our view templates inside of this layout

Creating topChart Template

Create a new file called topChart.html under client > views




```
<template name="topChart">
   <nav class="navbar navbar-default">
      <div class="container-fluid">
         <div class="text-center" id="navTitle">
            <strong>Top Charts</strong>
         </div>
      </div>
   </nav>
</template>
```

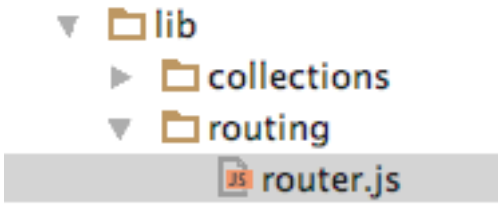
Here we define a template named “topChart”

Here we utilize the bootstrap navbar class to give us the App Store top bar

So now we have a top nav bar, but we still don’t have a data context containing the apps we want to display…. or a way of navigating to our topChart template.

Adding the topChart route

- At the beginning of this lesson we added a package called “iron:router”. Using this package, we can both provide a path for the user to access the topChart template and we can set the data context of that template to contain our topChart apps. Under lib > routing we will add a new file called router.js



Adding the topChart route continuted.

```
Router.configure({
    layoutTemplate: "masterLayout"
});
Router.route('/', {
    name: 'topChart',
    waitOn: function() {
        Meteor.subscribe('apps', {sort: {avgRating: -1, app_id: -1}, limit: 50});
    },
    data: function () {
        return {
            apps: Apps.find({}, {sort: {avgRating: -1, app_id: -1}, limit: 50})
        };
    }
});
```

Here we tell iron router to render our individual templates inside of our masterLayout template

Here we describe the route path. In this case, the path is simply '/' which means we are defining the root path of our web app

The template this route will useß is our topChart template (Meteor can infer this from the 'name' property'

Iron Router will hold off rendering our page until the "waitOn" function completes. We are returning a subscription to our "apps" publication that we set up earlier. We're passing a set of options to the server, telling the server to sort our apps based off of avgRating and app_id. And to only give us a max of 20 results.

The return value of the data function becomes the Blaze template's "data context". Here we are returning an object with a single property "apps" that contains all the Apps returned via our subscription.

Updating the topChart Template

Now that we've created our route, we should be able to view the current topchart template at http://localhost:3000. Let's update the template to utilize our new data context.

```
<template name="topChart">
  <nav class="navbar navbar-default">
    <div class="container-fluid">
      <div class="text-center" id="navTitle">
        <strong>Top Charts</strong>
      </div>
    </div>
  </nav>
  <ul class="list-group">
    {{#each apps}}
      {{> appPreview}}
    {{/each}}
  </ul>
</template>
```

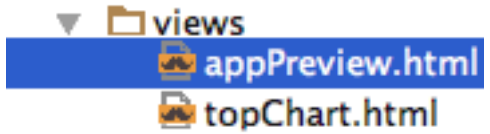
Here we utilize the built in Spacebars {{#each}} function to iterate through the "apps" property that we set inside our route's "data" function.

Inside the {{#each}}{{/each}} block the data context changes from "apps" to be the app in scope for each iteration.

Here we call the "appPreview" template (which we have yet to define). Since it is inside the {{#each}} block it will be passed the data for each individual app as its context as we iterate through our parent data context.

Creating the appPreview template

Create a new file called appPreview.html under client > views




```
<template name="appPreview">
  <li class="list-group-item">
    <div class="row">
      <div class="col-xs-1 appRank text-muted">
      </div>
      <div class="col-xs-3 appIconPreview">
        <img height=50 width=50 src="{{iconUrl}}"/>
      </div>
      <div class="col-xs-6 nameColumn">
        <a href="{{pathFor 'appPage'}}">{{name}}</a><br/>
        <div style="display:flex">
          {{> starsRating rating=avgRating class='mystar' size='sm'}}
        </div>
      </div>
      <div class="col-xs-2 text-right getApp">
        <a href="{{pathFor 'appPage'}}" class="btn btn-primary">+ Get</a>
      </div>
    </div>
  </li>
</template>
```

Each app preview will be wrapped in an <li> element, as the topChart template will insert each app into a

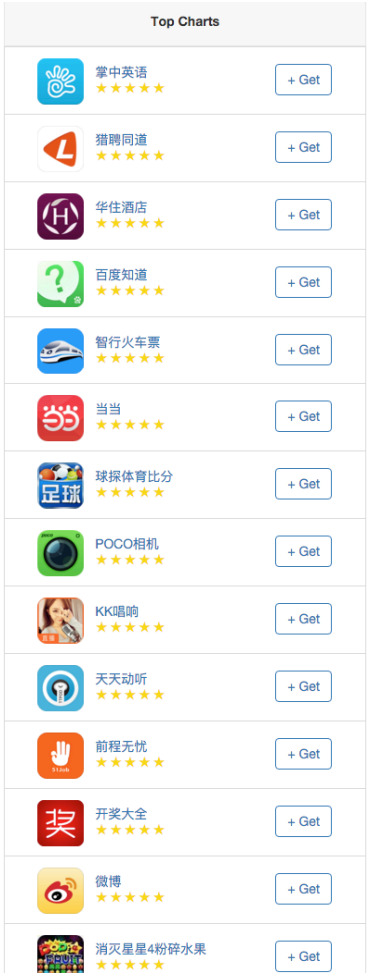
Here we set the img src to the iconUrl property that we scraped

The stars-rating package we installed earlier came with this template. We pass our apps avgRating property to the templates "rating" property so that it can color the correct # of stars.

pathFor is a built in helper that takes the name of a route and returns the correct path. We haven't added the "appPage" route yet so we'll come back to this in a few slides.

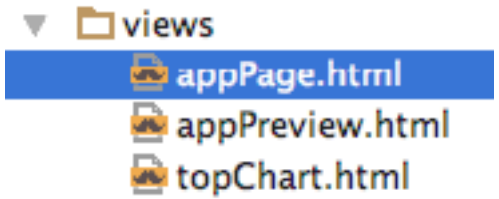
Creating Top Chart View

- Our topChart template should now iterate through the “apps” passed to it by Iron Router. If you visit http://localhost:3000 your app should now look like below:



Creating the appPage Template

We have one more template to make in order to show the app's details page. Create a file named appPage.html under client > views




```
<template name="suggestedApp">
  <div class="reccomendedApp">
    <a href="{{pathFor 'appPage'}}"><img height=50 width=50 src="{{iconUrl}}"/></a>
    <a href="{{pathFor 'appPage'}}" style="display:block;">{{name}}</a>
  </div>
</template>

<template name="appPage">
  <nav class="navbar navbar-default">
    <div class="container-fluid">
      <div class="navbar-brand">
          <a id="backLink"><span  class="glyphicon  glyphicon-chevron-left" aria-hidden="true"></span>
      </div>
    </div>
  </nav>
  <div class="appPageContent">
    <div class="row">
      <div class="col-xs-5">
        <img height=120 width=120 src="{{iconUrl}}"/>
      </div>
      <div class="col-xs-7 appInfo">
        <div class="appTitle">{{name}}</div>
        <p class="text-muted">{{developer}}</p>
        <div class="ratingsArea">
          {{> starsRating rating=avgRating class='mystar' size='sm'}}
        </div>
      </div>
    </div>
    <hr/>
```

Until now we've only defined one template per html file. By using a separate template for the suggested apps, we can set the data context for each suggested app and access it's properties.

This is our "back button" on the NavBar. We'll look at how the click listener is attached to this element in a couple slides.

Similar to our app_preview template, we use some of the properties from our App that have been stored in the template'

Creating the appPage Template cont.

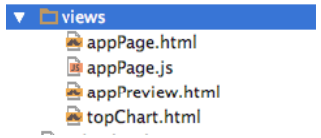
```
{{#if reccomendedApps}}
        <div class="row">
          <div class="col-xs-12">
            Customers also like
          </div>
        </div>
        <div class="row overflowWrapper">
          <div class="suggestedAppContainer">
            {{#each reccomendedApps}}
              {{> suggestedApp getSuggestedApp this}}
            {{/each}}
          </div>
        </div>
        <hr/>
      {{/if}}
      <div class="row">
        <div class="col-xs-12">
          <strong>Description</strong>
        </div>
      </div>
      <div class="row" style="margin-top:10px;">
        <div class="col-xs-12">
          <p class="text-muted">
            {{description}}
          </p>
        </div>
      </div>
    </div>
</template>
```

Since the reccomendedApps property is itself an array of app_ids we can iterate through the list passing each app_id to the suggestApp template.

The problem is that what we have are app_ids, not the actually App object from our collection that we need to set the data context of the suggestApp template. We will pass this to a template "helper" function called getSuggestedApp. More on this in the following slide.

Adding helper functions/event listeners

Similar to how we attached a data context to our blaze templates, we can also attach helper functions and event listeners. Create a new file called appPage.js under client > views




```
Template.appPage.helpers({
   getSuggestedApp: function(appId) {
      Meteor.subscribe('singleAppByAppId', appId);
      return Apps.findOne({app_id: appId});
   }
});

Template.appPage.events({
   "click #backLink" : function(evt) {
      history.back();
   }
});
```

Template.{{templateName}}.helpers() takes a JSON object where the keys (like getSuggestedApp) are the name of the helper function and the value is the function. These helper functions can be called directly from a blaze template like we saw on the previous slide.

The app_id string is passed to the helper function as a parameter. We utilize our singleAppByAppId subscription. We then use Apps.findOne() to actually retrieve the app Object and return it to our Blaze template. This object is then passed to our suggestedApp template as the data context

Similar to the helpers feature, you can attach JQuery-style event listeners on a template by passing a JSON object to Template.{{templateName}}.events(). The key here is the event type (click in this case) followed by the CSS selector (#backLink in this case). The values are the functions to be executed on event click. Here we are using a feature of iron router "history.back()" to bring us back to the previous page.

Adding the appPage Route

Let's update our router.js file that we created earlier under lib > routing

```
Router.configure({
  layoutTemplate:"masterLayout"
});

Router.route('/', {
  name: 'topChart',
  waitOn: function() {
    Meteor.subscribe('apps', {sort: {avgRating: -1, app_id: -1}, limit: 50});
  },
  data: function () {
    return {
      apps: Apps.find({}, {sort: {avgRating: -1, app_id: -1}, limit: 50})
    };
  }
});

Router.route('/app/:_id', {
  name: 'appPage',
  waitOn: function() {
    Meteor.subscribe('singleApp', this.params._id);
  },
  data: function () {
    return Apps.findOne(this.params._id);
  }
});
```

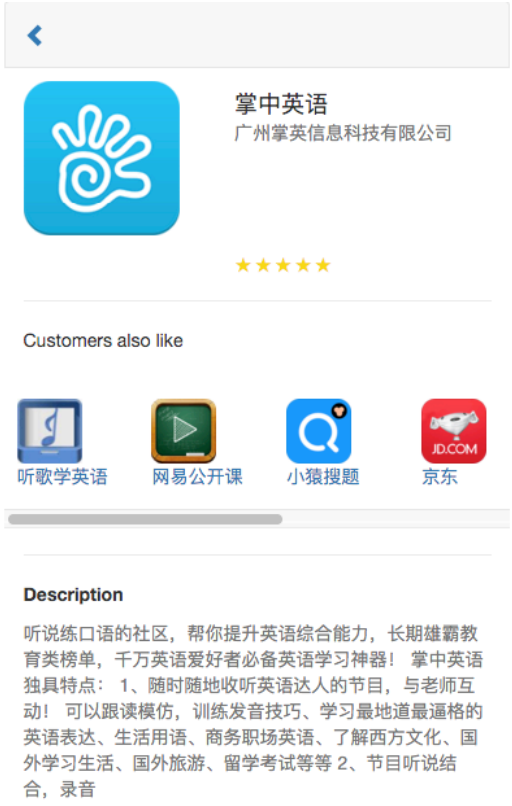
Our new path will be /app/ followed by an app's id. The ':' tells Iron Router that this is a variable that will be bound to the _id parameter.

This waitOn function only subscribes to the 'singleApp' publication, passing the app id passed in the URL by using this.params._id.

Bind the app with the given id to the data context of our template.

Tying It All Together

Now those 'pathFor' helpers we used in our appPreview template will automatically produce links to the individual appPage's for each app in our list. Clicking on the +Get button or title for any apps on the Top Chart list should now bring us to that app's details page. We can even navigate App to App by clicking on our suggested apps!



Student Project & Resources

Project Set Up

1. Download Git: https://git-scm.com/downloads
2. Clone repo: git clone https://github.com/BitTiger/meteorjs_app_store.git
3. Download NodeJS - https://nodejs.org/en/
4. Download MeteorJS - https://www.meteor.com/
   Tip: Sign up for a Meteor account at the end of the installation process.
5. Download a text-editor/IDE of choice.
   - Atom Text Editor
   - Webstorm IDE
   - Sublime Text Editor

Meteor Resources

- Meteor Documentation: http://docs.meteor.com/#/full/selectors (with direct link for more info on query selectors). Also http://docs.meteor.com/#/full/mongo_collection for info on Collection insert/update/remove functions.
- If you want to go deeper into Meteor, there's a great free tutorial on the Meteor site.
- If you want to go even deeper, I highly suggest Discover Meteor (an ebook + project to follow for $30).

Project Challenge

Earlier we discussed that since many apps share the same avgRating (number of stars), we had to use the app_id as a secondary sorting mechanism.

Earlier we added a property in fixtures.js called “numberOfRecommendations”. This property should be calculated by tallying the number of times this app appeared in another app’s recommended apps list. We will add our new logic in fixtures.js side our main if{} block right after we finish loading out data.

To achieve this:

1. Query the Meteor Apps Collection to retrieve all apps in MongoDB – convert the collection cursor into an **array**.
2. Iterate through the Apps array.
3. For each App, iterate through its reccomendedApps array (**if it exists**).
4. Use each app_id to update the appropriate MongoDB record by incrementing the numberOfRecommendations property.
5. Display our new property in the appPage template.

Tip: Our code will only run once since it’s wrapped an If{} block checking for an empty App collection. To reset the database, stop meteor (control + c in the console) and type ’meteor reset’.

Tiger AppStore

--Web in Java

Qun Wu

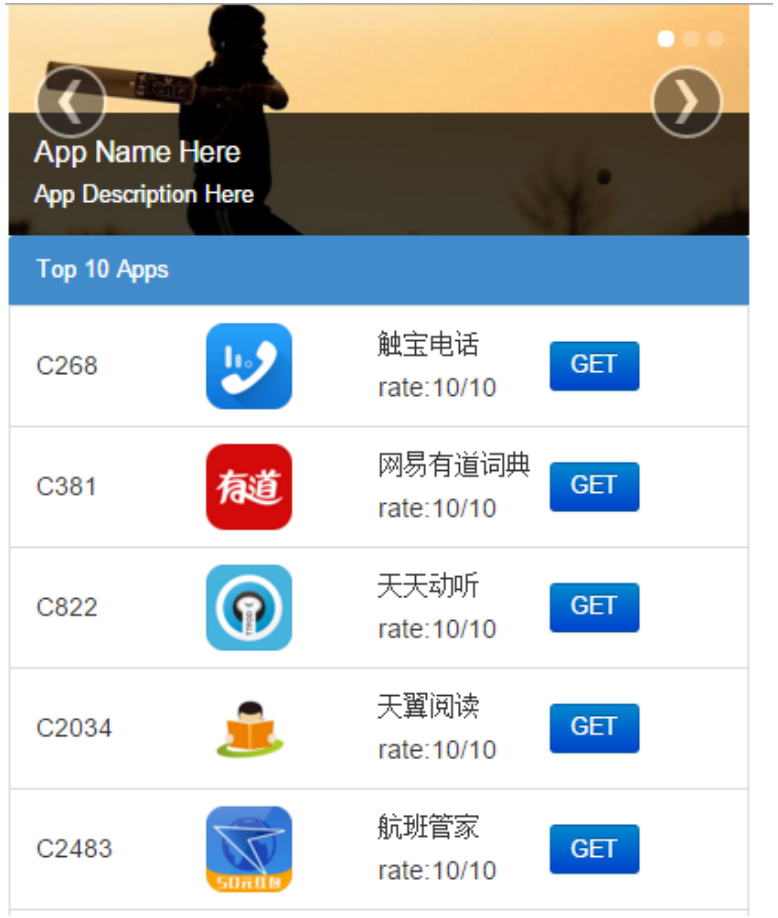
Content

- Project scope and goal
- Techniques used
- Web Architecture
- Data Source
- Problems
- Further work
- Reference
- Web Components—Coding Practice

Project scope and goal

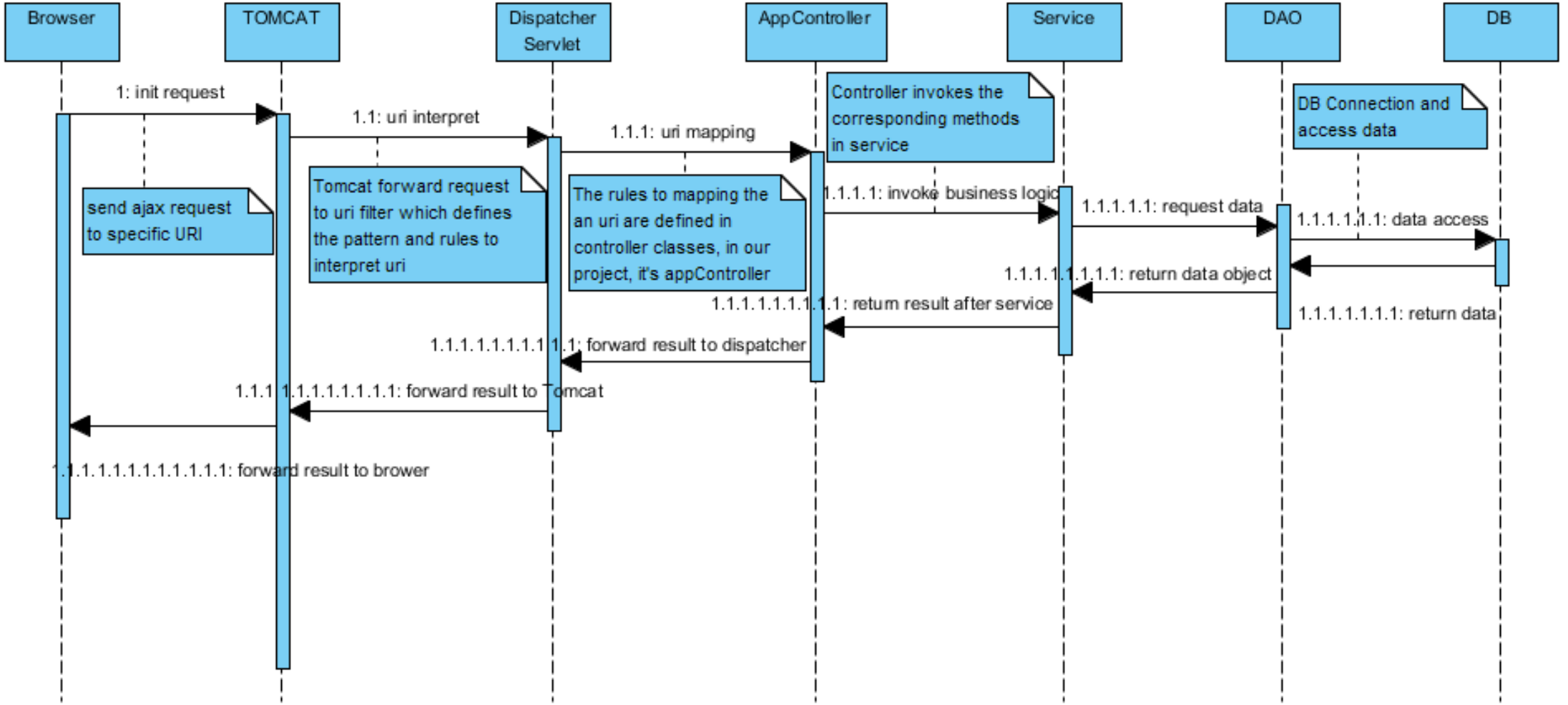
- Problem Domain: develop a web system to show apps and relative recommendation.
- Problem scope: 1. server/running environment setup,
  2. database, web services, front web pages
  → full stack project
- Use cases: 1. initially, show top 10 popular apps
  2. click “Get” button to obtain the app detail

Project - view of project



Project – Web Behavior

- Web Project General Behavior



Technique used- Front-end

- Front-end pages:

HTML, CSS(bootstrap), JS(jQuery, AngularJS)

1. Import JQuery before bootstrap lib
2. AngularJS is used to manipulates DOM tree like what JQuery can do, but it does not depend on JQuery. AngularJS has its own implementation to changes DOM tree, called JQLite
3. AngularJS has ‘http’ service to send Ajax request

Technique used-Back-end

- Back-end(JAVA web, Restful API):

Control layer: Spring-mvc

Business layer: JAVA

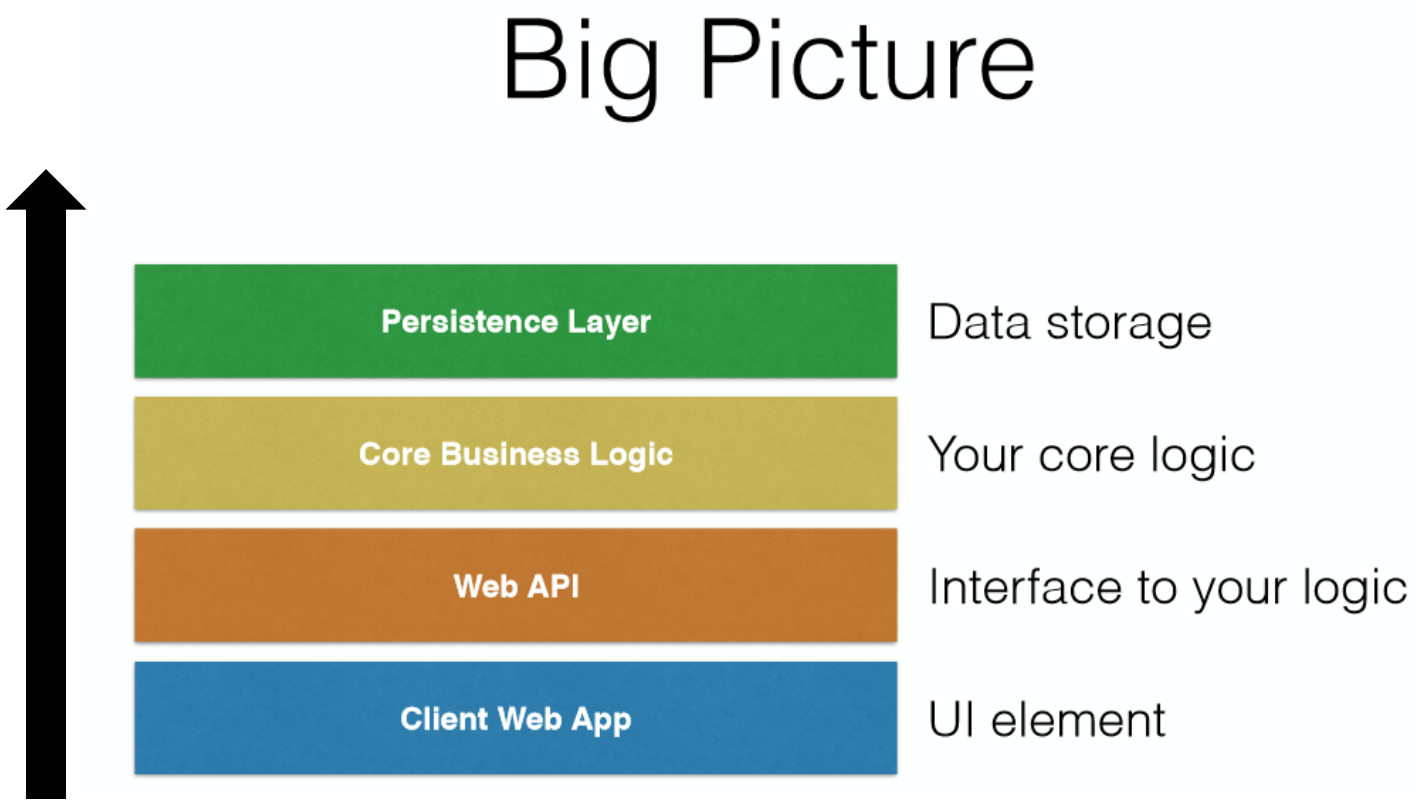
DB Connection & Operation: Hibernate

Spring framework throughout the whole back-end(be introduced later)

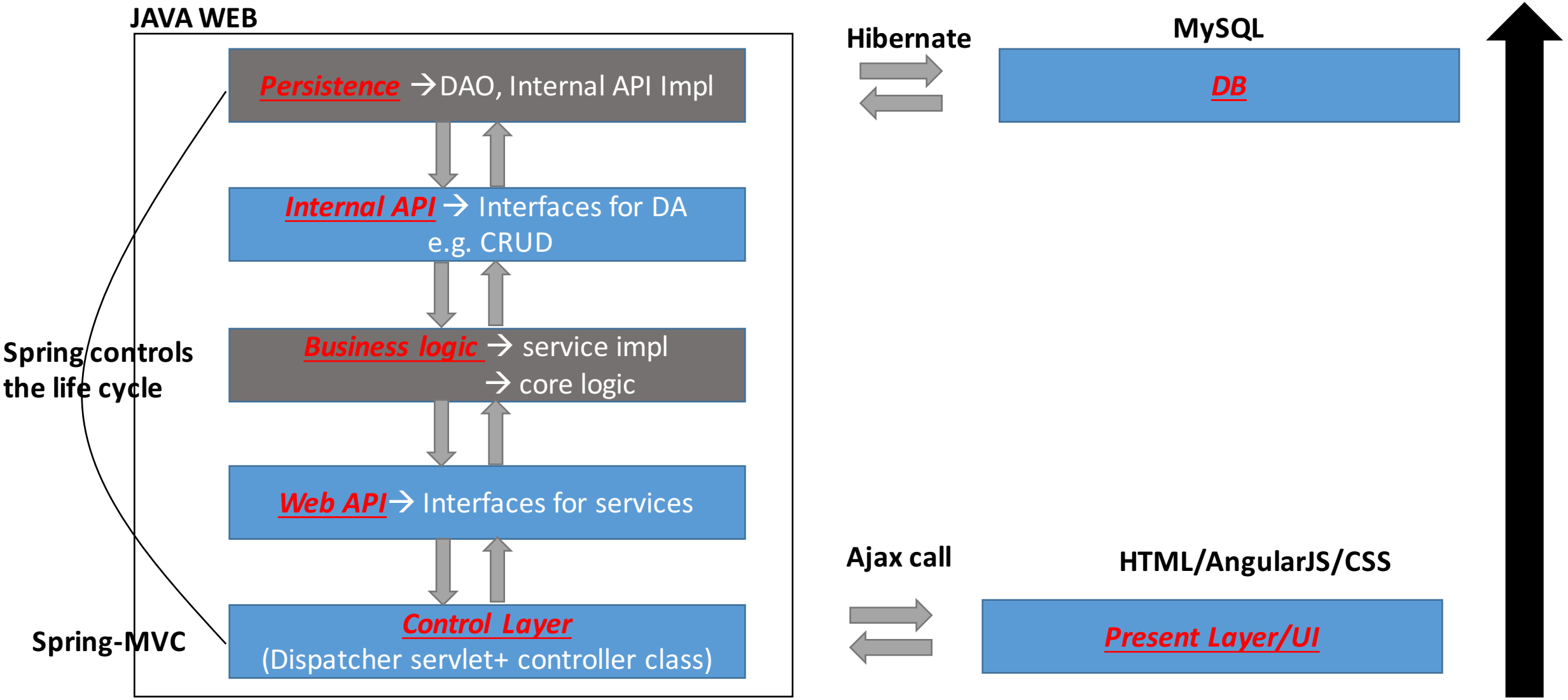
- Database: MySQL, not MongoDB

Web Architecture

- Service-oriented-architecture (SOA)



Web Architecture-Real architecture:



Web Component – Spring

- Java bean generation/ Instantiation of object(IOC)

Recommend: **setter** Injection (Jackson also use it). Also has constructor injection.

- Design pattern e.g. singleton, prototype.

Singleton vs Prototype:

e.g. List<User> users ;→every time after login, add user to list.

If “users” is

Singleton: one list “users”, keeps all users

Prototype: users will be initiated newly each time

- Java web frameworks integration depends on spring. Why not hibernate, struts? → java bean support

Web Component – Spring - Injection

- Which parts in project shall be initiated by Spring?

Java beans → service and dao

e.g. AppService appServ = new AppServImpl();

Spring→ Object of 'AppServImpl' to a JAVA BEAN with id 'appServBean'

→ 'appServBean' will be injected to variable "appServ" by Spring

- How about the entities.

e.g. When user login → encapsulate username and pwd into object

Jackson → encapsulate data from json into object

Hibernate → encapsulate data from MySQL into object

Data Source

- Storage Platform: MySQL 5.5

Download: http://dev.mysql.com/downloads/mysql/

(→ page partial update when select download options)

Installation:
http://jingyan.baidu.com/article/ed2a5d1f4968c909f6be179f.html

- Data Source:

Import sql file into MySQL directly

```
mysql> source file_name
```

http://dev.mysql.com/doc/refman/5.0/en/mysqlimport.html

Problems

- GET/delete method do **not** accept **JSON** DATA in request
- Spring-mvc Jackson: 500 internal error/400 bad request

1. How Jackson gets values from json data or object fields

→Getter and setter

2. What happen when null value appears in object or json data

→400 bad request when send json to server

→500 internal error when server send json to front-end

3. MySQL/mongodb Chinese language confict

→Code page 936 to fix the problem in mongodb

→In mysql it cannot be fixed. But it does not have impact on project. Pages work well

Further work

- Do not use SQL, intead, use mongodb directly

Sol-1: hibernate OGM (important: still support HQL, also support native query in JPA)

Sol-2: do not use hibernate either. Instead, use java code

- Enlarge the scope of project.

1. User management
2. Add category for app
3. Add comments

- Make it be a real single page website with AngularJS

Sol: make HTMLs be independent components of one page, dynamically load these components into single page

Reference

- IOC:

https://zh.wikipedia.org/wiki/%E6%8E%A7%E5%88%B6%E5%8F%8D%E8%BD%AC

- AOP: http://baike.baidu.com/subview/73626/13548606.htm
- MongoDB with JAVA, demo : http://www.cnblogs.com/hoojo/archive/2011/06/02/2068665.html
- MongoDB with JAVA, doc :

https://docs.mongodb.org/getting-started/java/

- Hibernate Doc:

http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html_single/

- Hibernate OGM HQL&JP-QL support

http://docs.jboss.org/hibernate/ogm/4.2/reference/en-US/html/ch07.html

http://docs.jboss.org/hibernate/ogm/4.2/reference/en-US/html/ch07.html

- Running environment setup

- Start to install MySQL (5.5), Tomcat (8.0)
- import data
- Connect Tomcat with your IDE (MyEclipse)

about 15 mins

Web Component – create a web project

- Create a web project (add maven support → optional)
- Correct web.xml

1. <welcome-file-list>
2. <servlet-name>+<servlet-class>→url filter, depends on control layer
3. <servlet-name>+<url-pattern>/</url-pattern>→url pattern to match

- Import jar files for spring-mvc, spring framework, hibernate

Web Component – create a web project

- Entity Definition
- Code Web service, including DAO layer which uses hibernate support

1. Web API + service implementation
2. Internal API + DAO(crud first, then extend others functions)

- Configure the spring-config.xml (dynamic change, read by web.xml)

1. Generate java beans, control their left cycle
2. Integrate Hibernate

Web Component - front-end

- Front-end pages:
- Import jQuery lib
- Import angularJS lib
- Import bootstrap lib and necessary style file
- Quickly generate a web layout

http://www.runoob.com/try/bootstrap/layoutit/