17 Jan 2015

Web App Architecture - the Spring MVC - AngularJs stack

Spring MVC and AngularJs together make for a really productive and appealing frontend development stack for building form-intensive web applications.

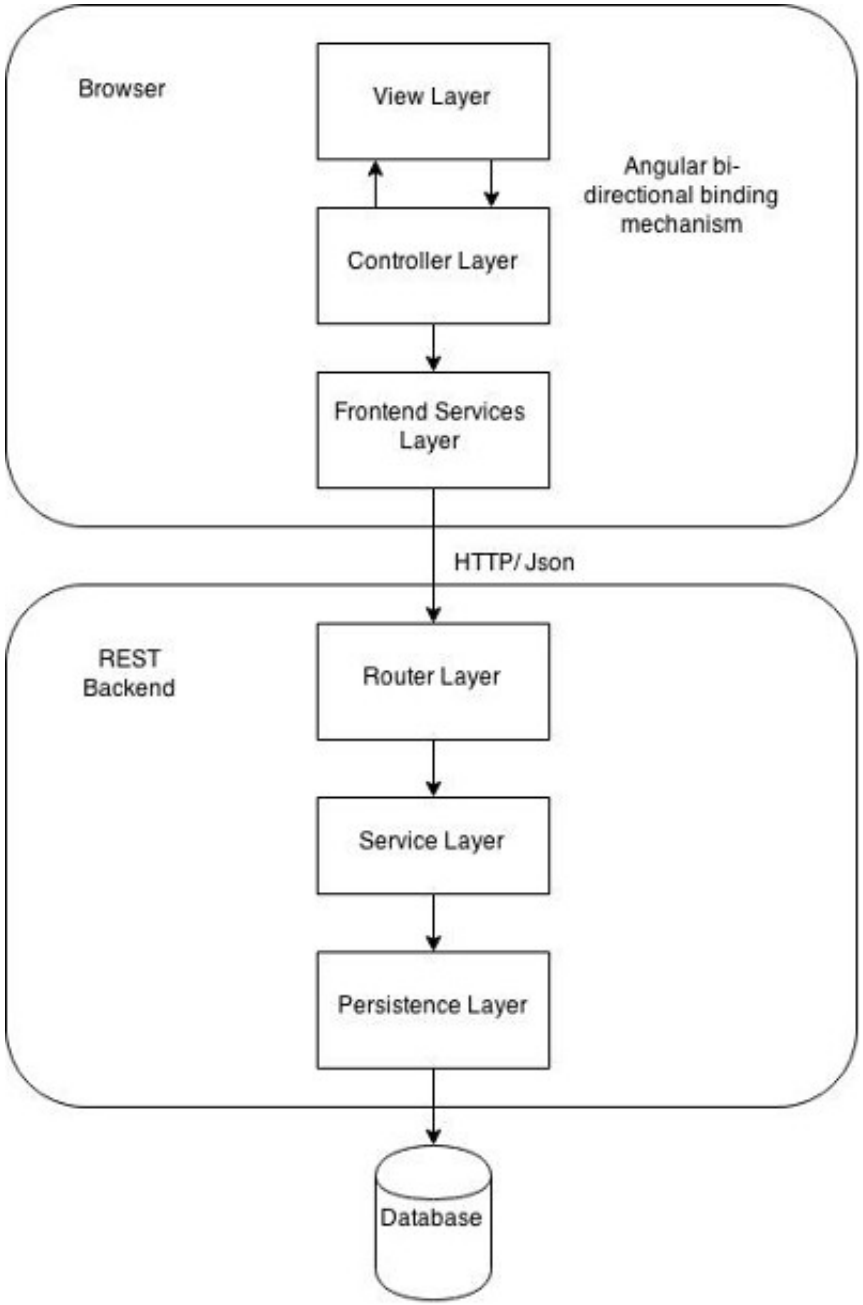
In this blog post we will see how a form-intensive web app can be built using these technologies, and compare such approach with other available options. A fully functional and secured sample Spring MVC / AngularJs web app can be found in this github repository.

We will go over the following topics:

- The architecture of a Spring MVC + Angular single page app
- How to structure a web UI using Angular
- Which Javascript / CSS libraries complement well Angular?
- How to build a REST API backend with Spring MVC
- Securing a REST API using Spring Security
- How does this compare with other approaches that use a full Java-based approach?

The architecture of a Spring MVC + Angular single page web app

Form-intensive enterprise class applications are ideally suited for being built as single page web apps. The main idea compared to other more traditional server-side architectures is to build the server as a set of stateless reusable REST services, and from an MVC perspective to take the controller out of the backend and move it into the browser:



The client is MVC-capable and contains all the presentation logic which is

separated in a view layer, a controller layer and a frontend services layer. After the initial application startup, only JSON data goes over the wire between client and server.

How is the backend built?

The backend of an enterprise frontend application can be built in a very natural and web-like way as a REST API. The same technology can be used to provide web services to third-party applications - obviating in many cases the need for a separate SOAP web services stack.

From a DDD perspective, the domain model remains on the backend, at the service and persistence layer level. Over the wire only DTOs go by, but not the domain model.

How to structure the frontend of a web app using Angular

The frontend should be built around a view-specific model (which is not the domain model), and should only handle presentation logic, but no business logic. These are the three layers of the frontend:

The View Layer

The view layer is composed of Html templates, CSS, and any Angular directives representing the different UI components. This is an example of a simple view for a login form:

Log In

New user?

The Controller Layer

The controller layer is made of Angular controllers that glue the data retrieved from the backend and the view together. The controller initializes the view model and defines how the view should react to model changes and vice-versa:

```
angular.module('loginApp', ['common',  'editableTableWidgets'])
    .controller('LoginCtrl', function ($scope, LoginService) {

        $scope.onLogin = function () {
            console.log('Attempting login with username ' + $scope.vm.userna
me + ' and password ' + $scope.vm.password);
```

```
            if ($scope.form.$invalid) {
                return;
            }

            LoginService.login($scope.vm.userName, $scope.vm.password);

        };

    });
```

One of the main responsibilities of the controller is to perform frontend validations. Any validations done on the frontend are for user convenience only - for example they are useful to immediately inform the user that a field is required.

Any frontend validations need to be repeated in the backend at the service layer level due to security reasons, as the frontend validations can be easily bypassed.

The Frontend Services Layer

A set of Angular services that allow to interact with the backend and that can be injected into Angular controllers:

```
angular.module('frontendServices', [])
    .service('UserService', ['$http','$q', function($http, $q) {
        return {
            getUserInfo: function() {
                var deferred = $q.defer();

                $http.get('/user')
                    .then(function (response) {
                        if (response.status == 200) {
                            deferred.resolve(response.data);
                        }
                        else {
                            deferred.reject('Error retrieving user info');
                        }
                });

                return deferred.promise;
```

```
}
```

Let's see what other libraries we need to have the frontend up and running.

Which Javascript / CSS libraries are necessary to complement Angular?

Angular already provides a large part of the functionality needed to build the frontend of our app. Some good complements to Angular are:

- An easily themeable pure CSS library of only 4k from Yahoo named PureCss. Its Skin Builder allows to easily generate a theme based on a primary color. Its a BYOJ (Bring Your Own Javascript) solution, which helps keeping things the 'Angular way'.

- a functional programming library to manipulate data. The one that seems the most used and better maintained and documented these days is lodash.

With these two libraries and Angular, **almost any form based**

application can be built, nothing else is really required. Some other libraries that might be an option depending on your project are:

- a module system like requirejs is nice to have, but because the Angular module system does not handle file retrieval this introduces some duplication between the dependency declarations of requirejs and the angular modules.

- A CSRF Angular module, to prevent cross-site request forgery attacks.

- An internationalization module

How to build a REST API backend using Spring MVC

The backend is built using the usual backend layers:

- Router Layer: defines which service entry points correspond to a given HTTP url, and how parameters are to be read from the HTTP request

- Service Layer: contains any business logic such as validations, defines the scope of business transactions

- Persistence Layer: maps the database to/from in-memory domain objects

Spring MVC is currently best configured using only Java configuration. The `web.xml` is hardly ever needed, see here an example of a fully configured application using Java config only.

The service and persistence layers are built using the usual DDD approach, so let's focus our attention on the Router Layer.

The Router Layer

The same Spring MVC annotations used to build a JSP/Thymeleaf application can also be used to build a REST API.

The big difference is that the controller methods do not return a String that defines which view template should be rendered. Instead the `@ResponseBody` annotation indicates that the return value of the controller

method should be directly rendered and become the response body:

```
    @ResponseBody
    @ResponseStatus(HttpStatus.OK)
    @RequestMapping(method = RequestMethod.GET)
    public UserInfoDTO getUserInfo(Principal principal) {
        User user = userService.findUserByUsername(principal.getName());
        Long todaysCalories = userService.findTodaysCaloriesForUser(principa
l.getName());

        return user != null ? new UserInfoDTO(user.getUsername(), user.getMa
xCaloriesPerDay(), todaysCalories) : null;
    }
```

If all the methods of the class are to be annotated with `@ResponseBody`, then it's better to annotate the whole class with `@RestController` instead.

By adding the Jackson JSON library, the method return value will be directly converted to JSON without any further configuration. Its also

possible to convert to XML or other formats, depending on the value of the `Accept` HTTP header specified by the client.

See here an example of a couple of controllers with error handling configured.

How to secure a REST API using Spring Security

A REST API can be secured using Spring Security Java configuration. A good approach is to use form login with fallback to `HTTP Basic` authentication, and include some `CSRF` protection and the possibility to enforce that all backend methods are only accessible via `HTTPS`.

This means the backend will propose the user a login form and assign a session cookie on successful login to browser clients, but it will still work well for non-browser clients by supporting a fallback to HTTP Basic where credentials are passed via the `Authorization` HTTP header.

Following OWASP recommendations, the REST services can be made minimally stateless (the only server state is the session cookie used for

authentication) to avoid having to send credentials over the wire for each request.

This is an example of how to configure the security of a REST API:

```
http
        .authorizeRequests()
        .antMatchers("/resources/public/**").permitAll()
        .anyRequest().authenticated()
        .and()
        .formLogin()
        .defaultSuccessUrl("/resources/calories-tracker.html")
        .loginProcessingUrl("/authenticate")
        .loginPage("/resources/public/login.html")
        .and()
        .httpBasic()
        .and()
        .logout()
        .logoutUrl("/logout");
```

```
        if ("true".equals(System.getProperty("httpsOnly"))) {
            LOGGER.info("launching the application in HTTPS-only mode");
            http.requiresChannel().anyRequest().requiresSecure();
        }
```

This configuration covers the authentication aspect of security only, choosing an authorization strategy depends on the security requirements of the API. If you need a very fine-grained control on authorization then check if Spring Security ACLs could be a good fit for your use case.

Let's now see how this approach of building web apps compares with other commonly used approaches.

Comparing the Spring / MVC Angular stack with other common approaches

This approach of using Javascript for the frontend and Java for the backend makes for a simplified and productive development workflow.

When the backend is running, no special tools or plugins are needed to achieve full frontend hot-deploy capability: just publish the resources to

the server using your IDE (for example hitting `Ctrl+F10` in IntelliJ) and refresh the browser page.

The backend classes can still be reloaded using JRebel, but for the frontend nothing special is needed. Actually the whole frontend can be built by mocking the backend using for example json-server. This would allow for different developers to build the frontend and the backend in parallel if needed.

Productivity gains of full stack development?

From my experience being able to edit the Html and CSS directly with no layers of indirection in-between (see here a high-level Angular comparison with GWT and JSF) helps to reduce mental overhead and keeps things simple. The edit-save-refresh development cycle is very fast and reliable and gives a huge productivity boost.

The largest productivity gain is obtained when the same developers build both the Javascript frontend and the Java backend, because often simultaneous changes on both are needed for most features.

The potential downside of this is that developers need to know also Html, CSS and Javascript, but this seems to have become more frequent in the last couple of years.

In my experience, going full stack allows to implement complex frontend use cases in a fraction of the time than the equivalent full Java solution (days instead of weeks), so the productivity gain makes the learning curve definitely worth it.

Conclusions

Spring MVC and Angular combined really open the door for a new way of building form-intensive web apps. The productivity gains that this approach allows make it an alternative worth looking into.

The absence of any server state between requests (besides the authentication cookie) eliminates by design a whole category of bugs.

For further details have a look at this sample application on github, and let us know your thoughts/questions on the comments bellow.

jhadesdev

A blog about tips and tricks on polyglot development.

Share this post