Editor: **Robert Blumen**
Salesforce
robert@robertblumen.com

Yakov Fain on Angular

Matthew Farwell

From the Editor

Episode 293 of Software Engineering Radio, with guest Yakov Fain, is about Angular, the popular web development framework. The edited portion of the interview presented here covers the basics and primary use cases. The portions omitted here for reasons of space cover working with streams, asynchronous programming, extensibility, Angular's roots in the TypeScript variant of JavaScript, testing, and performance. To hear the full interview, visit se-radio.net or access our archives via RSS at feeds.feedburner.com/se-radio. —*Robert Blumen*

Matthew Farwell: Yakov Fain is coauthor of *Angular 2 Development with TypeScript* and a software architect at the IT consultancy Farata Systems. Yakov, what is Angular?

Yakov Fain: If you had asked me this question two years ago, I'd have said it is a framework for web development. But now, I'd rather say this is a platform for development of the front end of web applications, mobile applications, and desktop applications. It is developed by Google. It was announced as a new version of the superpopular AngularJS framework, but actually it's a complete rewrite. It's a different framework, and much more powerful than it used to be.

Is Angular a framework or a library, and what's the difference?

It's definitely a framework. The difference is that you can combine a library with other libraries. You decide on the architecture of the application; you can write everything in pure JavaScript plus library XYZ.

A library doesn't prescribe how to structure your applications. A framework does. You insert your code while playing by certain rules.

I've heard that distinction called "the Hollywood principle": "Don't call us; we'll call you." Do you agree?

The Hollywood principle is often mentioned in regard to one aspect of the framework: dependency injection. "Don't call me; I'll call you" means that I'm not going to be creating instances of the object whenever I want. The container—in this case, Angular—would do it for me and insert [the instance] where I want it to go.

But Angular is much more than dependency injection. It has a superpowerful router. It comes with the library RxJS, for reactive programming. Also, there is a module for server-side rendering. There is integrated bundling tooling. It comes with the tools that allow you to create bundles for deployment to webservers. You could say that the Angular framework comes with the batteries included.

Why do we need a JavaScript development framework?

That's not a difficult question to answer: because not every web application developer is supersmart. Not everyone can pick and choose a la carte. Not everyone can figure out "Which library I need, or do I need them?" or "Maybe I just will write everything in JavaScript."

If you are a one-man shop and a very experienced developer, maybe you don't need a framework. However, we work, in many cases, in teams. The skill level within any team, especially in the enterprise world, varies. Having a framework greatly simplifies the understanding of a particular application. Say I am a JavaScript developer who also knows Angular (or any framework), and I join a team. If you hired me knowing that I know this framework, you know that I will have an easier time getting on board and understanding how your application works, because it plays by certain rules and has a certain structure.

0740-7459/17/$33.00 © 2017 IEEE

SOFTWARE ENGINEERING RADIO

Visit www.se-radio.net to listen to these and other insightful hour-long podcasts.

RECENT EPISODES

- 301—Jason Hand of Victor Ops tells host Bryan Reinero how to handle outages in cloud-based systems—and what we can learn from them.
- 302—Host Kim Carter has a far-reaching dialogue with Haroon Meer on what software engineers must know about network security.
- 303—Host Felienne talks to Zachary Burt about freelancing as a career.

UPCOMING EPISODES

- 306—Ron Lichty, author of *Managing the Unmanageable*, talks with host Nate Black about managing software engineers.
- 307—Harsh Sinha delves into software product management, with host Bryan Reinero.
- 308—Gregor Hohpe tells host Bryan Reinero the 37 things enterprise architects must know.

Would you describe Angular as an opinionated framework?

Definitely.

What's an opinionated framework?

A framework that has an opinion of how you are supposed to do things. In many cases, Angular places restrictions on how to do things. Suppose you want to implement an Angular application (which in most cases is a so-called single-page application, where the browser doesn't refresh the entire page but only changes certain fragments on the page as the user navigates). There is a router that tells you how you are supposed to navigate from point A to point B. There is a certain API that you have to stick to.

What are Angular's goals, then?

The first goal of Angular is to bring common-ground web application development. The other goal is to give developers tools that will allow them to easily solve typical issues that any application needs to solve. For example, "How do I navigate the page?," "How do I build the project?," "How do I make sure that the user cannot click on a link until certain data is available?," and "How do I modularize the application?" My application is not a monolith; it's cut into pieces. Smaller pieces are loaded—and some pieces are loaded on demand.

What are the parts of an Angular application?

That's a good question because Angular—compared to AngularJS—made a complete switch from being a model–view–controller framework to a component-based framework. The application is a bunch of components, starting from the top-level component, known as the *root*, which may have child components. Children may have their own children, and so on. The application is a bunch of Legos—components that can communicate with each other. Component A can pass data to component B. Component B can send events describing things that happen inside itself. Components can be brought together in a loosely coupled fashion—which is super important.

A component is a piece of code with certain functionality. The recommended language is TypeScript, although you can also write in flavors of JavaScript such as ECMAScript 5, 6, or 7.

A component consists of a file, which has classes or scripts in TypeScript. You can put HTML in another file, or you may keep it inline. The third file could contain Cascading Style Sheets (CSS). It is also recommended to include testing, which you can do by creating a file with the extension spec.ts. This group of files represents a component. Typically, these files are in one directory. A project will consist of multiple directories with subdirectories. Each directory will have a component.

Is a component visual?

Yes. Think of it this way: You write a class in TypeScript. You add a decorator or the annotation `@component` and prepend it to the class definition. Inside that, you define the template; you define CSS. This is what makes a class a component. Without that `@component` annotation, it's not a component. If you want to implement a piece of logic, you create a class and put any logic you want there. But this is not a component.

Does Angular have any concept of a service?

Yes. A service is also a class. But it contains only some business logic such as communication with the

back end or calculation. For example, a product service would fetch data about products. Services are injected into components. If you write a component, it has a constructor, which Angular uses for injection.

Does Angular help you with the bits that aren't visual, such as common code between all the components?

Angular comes with an excellent tool called Angular CLI (command line interface) that allows you to quickly generate a component, directive, class, or service.

Then, in the constructor, if you define an argument whose type is the class of one of your services, Angular is smart enough to understand: "Oh, he wants a product service. Let me create an instance and inject it into this component."

How do I put an Angular application together?

Starting up is pretty painless. A year ago, many people complained that Angular 2 was not easy to get up and running because it comes with a whole bunch of configuration files and tooling, because you write it in TypeScript but it deploys in JavaScript, and because you need some kind of a tool to build the app.

So, the Angular team created Angular CLI. It can get you started quickly. You install it. It provides you with the command `ng`. Then you say `ng new`, and the name of the project, on the command line. In about 30 seconds it gives you a new project with a root directory, subdirectories, and everything you need. It's a minimalist application, kind of a "Hello world."

Either you open it up in your IDE or you can just run another command from the command line—for example, `ng serve`. This command instructs Angular CLI: "Build this app, create bundles out of it, and deploy it." You can open it in the browser, and it runs.

If you want to see the structure and work the way we do, you can pick any of the excellent available IDEs. I am using WebStorm from JetBrains. Another excellent IDE is Visual Studio Code (not to be confused with Visual Studio 2015), a free open source IDE. You open your IDE and you see a clean structure: folders, subfolders, the SRC folder, and the APP folder.

What is data binding?

Data binding is about keeping things synchronized. Say you have a field on the UI and a class variable in the TypeScript code. You want to make sure that, if somebody is entering something in the UI control on the webpage, this value will jump right into that variable. Or, if somebody modifies the value of that variable—say, you make a server request and the value changed—you want to immediately show it on the UI. You do this using data binding. You bind a variable in a class to a control on the UI. Or you can bind a button on the UI to a function that will handle that button-click event.

Is that two-way, or just one-way, from the model to the page?

Angular supports both ways, but the default is one-way. This is a big difference compared to AngularJS, where two-way binding was the default, but that caused some performance issues. So in Angular, they decided that unidirectional would be the default. But in some cases, two-way binding may be beneficial.

How you would handle things such as validation for forms?

Something called `NG model` provides syntax for binding in both directions. Angular CLI comes with a convenient and powerful forms API. Validation of forms is handled using the validation API.

Angular's forms API comes with two APIs: one is template-driven; the other one is called the reactive API. As you build the form, you can also attach zero or more validators to every form control. The thing will work automatically. The forms API also supports asynchronous validation. So, you can do validation on the server.

Suppose you want to validate the value of an e-mail. Angular 4 comes with a prepackaged validator for email. Or, you want to make a field required. Or, you want to do min or max allowed. These things come out of the box.

But you can also create a custom validation. For that, you can write a function. For example, in the US, we have social-security numbers: a unique ID for every person in the country. It must have nine digits, right? You would write a simple function containing a regular expression that would ensure that the input has nine digits.

How does reactive programming fit in?

Reactive programming in Angular is done with the support of the RxJS library. RxJS is built on the same principles as any reactive library: *observable* and *observer*. An observable is something that pushes data. An observer is an object or a function that knows what to do with the data when it arrives. [These things come] along with a bunch of operators that can handle the data en route, so to speak.

ABOUT THE AUTHOR

MATTHEW FARWELL is a senior software developer at Nexthink. Contact him at matthew@farwell.co.uk.

The data comes from observables to the subscriber, and you can chain a bunch of operators in there that process the data.

Angular in multiple places already gives you access to observable streams. When you move using a router from point A to point B, you may need to process the data. The screen with the product ID will show a list of products. From there, you want to go to another component, the one that shows product detail—but for which product? You need to pass an ID, right? In the router, you subscribe to changing ID values.

Are we talking about the two sides of a page?

Yes. Two components that can be sitting on the same page, like master–detail. The master component shows the list of products. You click on it to see the detail about [one product]. The detail component is separate.

With streaming, as you select different products, the stream gives you different IDs. As I mentioned already, in the forms, you can subscribe to observables. HTTP requests are handled using the same paradigm. When you make an HTTP request, you subscribe to an observable. You may use WebSocket to subscribe to incoming data the same way. The framework uses the same approach in many places.

So, an HTTP request is done via RxJS as an observable, and then eventually you get a response?

Right. HTTP is an asynchronous operation. An HTTP object comes with Angular, having the standard methods `GET`, `PUT`, `POST`, `DELETE`, and so on. Each of these methods returns an observable. When you see that in the documentation, it means that you can subscribe to it.

Moreover, it gives you a power you may not find in other places. When a user makes an HTTP request from a mobile phone in a slow-connection area, he or she becomes annoyed if the request doesn't come back within five seconds, so he or she makes another request. For example, "I wanted to check the price on the IBM in the stock market. No, no. It didn't come back. Let me check Apple's price." The first request never came back. However, it will come back eventually, and when it does, it will clog the bandwidth. Operators provide a way to kill the unfinished request in one line.

IEEE Software (ISSN 0740-7459) is published bimonthly by the IEEE Computer Society. IEEE headquarters: Three Park Ave., 17th Floor, New York, NY 10016-5997. IEEE Computer Society Publications Office: 10662 Los Vaqueros Cir., Los Alamitos, CA 90720; +1 714 821 8380; fax +1 714 821 4010. IEEE Computer Society headquarters: 2001 L St., Ste. 700, Washington, DC 20036. Subscribe to *IEEE Software* by visiting www.computer.org/software.

Postmaster: Send undelivered copies and address changes to *IEEE Software*, Membership Processing Dept., IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08854-4141. Periodicals Postage Paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Publications Mail Agreement Number 40013885. Return undeliverable Canadian addresses to PO Box 122, Niagara Falls, ON L2E 6S8, Canada. Printed in the USA.