Building modern enterprise applications from scratch: lessons learned

Dr. Clemens Wrzodek
Roche Diagnostics GmbH
Roche Innovation Center Penzberg

Keywords
Modern enterprise web-applications, SOFEA, Java, Angular, Oracle, Apex, backend, frontend, architecture

Introduction
Building a modern enterprise web-application from scratch holds many challenges, but also opportunities. A basic architecture seems simple: Oracle DB, Java backend and a Web-UI.

However, the landscape of available methodologies to realize the project has grown in the last years. Developers have many choices to develop their frontend, backend and database logic. JavaScript frameworks are mature, GWT offers a Java-based development of WebUIs, and Apex has many advantages that other frontend development platforms are missing. So what is the best choice here?
On the backend side, the application server itself is usually set in enterprise environments. Still, many choices and application server peculiarities remain: REST services tend to supersede SOAP and different ways to perform automated integration tests provide means to control stability of the whole project.

An Oracle Database can be called by the Java backend by using the Java Persistence API (JPA). However, there are situations in which this may not be the best choice. PL/SQL provides means to directly code business logic in Oracle databases and allows for exchanging Information with the Java backend by using Oracle Objects. What are the advantages or disadvantages of implementing the business logic in PL/SQL or Java? In which situations is a PL/SQL API superior to using JPA?

Architecture and setup

Most architectural designs of modern web-applications are rather simple and can be reduced to the database, a backend and a frontend. However, there are crucial differences in how the various parts are implemented. Traditionally, the backend layer is rather "thick" and intelligent, whereas the frontend and database layers are kept "dumb" and just display or store what the backend is redirecting. This is maybe due to the traditional frameworks, such as spring, struts, etc. However, in an increasingly service-oriented world, different design principles become more and more popular. One of them is SOFEA which stands for service oriented frontend architecture. Basically, this design principle guides architects to decouple the different tiers: let the client implement the application flow and MVC pattern using AJAX. The server should provide resources or services independently of the client's workflow. It should function as a central unit, processing business logic and providing access to the data layer. This design principle can also be applied in the other direction: Instead of duplicating available oracle database functionality in the backend tier, a PL/SQL API with oracle object types can easily be accessed from a backend and allows for using the full oracle database power. Oracle allows much more than just the storage and retrieval of data – cartridges, full-text-indices and the PL/SQL language are just some examples that can be used to further reduce backend complexity where needed.

How to implement the Web-UI

Many different technologies are available for building a modern web-ui: Apex, JSF, GWT, JavaScript, etc. There are many discussions of pros and cons of the different approaches. Probably all of them are good choices, however, there are some guidelines you can follow:

Apex is probably the way to go if you have experience in PL/SQL and you want to build small to medium-sized projects that mainly use reporting and grid-editing functionalities. The grids and reports built into Apex are powerful and outclass many hard-to-use JavaScript alternatives. From an architectural point of view, Apex is also your backend, so you can quickly build a Web-UI directly from a database table. However, that can also be considered a disadvantage. If you need a powerful backend where you implement a lot of business logic (beyond PL/SQL capabilities) and want to provide a public API, building a backend separately yields more opportunities. Technologies to write a dedicated frontend for a public SOAP/REST based backend are usually JSF/GWT or JavaScript (JS) with frameworks such as AngularJS or Ember. JSF is quite some time on the stage and easy to start for Java Developers. The code is usually not developed independently of the backend but coupled tightly. GWT is a promising JavaScript compiler initially developed by Google and transformed finally to an open-source project in 2013. GWT allows Java developers to quickly build JS-based applications. Due to its great integration into eclipse and the Java programming world, greater complexity can be handled effectively and large teams can use it to work on large projects. However, at the end of the day it is still JavaScript code that is deployed on the webserver and runs in the browser. Therefore, many great components today are released as JS/HTML5 apps and not necessarily directly as GWT component. GWT needs a wrapper to include those. So a more direct approach is developing the frontend with JS. JQuery allows a quick start for building small apps. However, building larger apps in JS, even with JQuery, can quickly lead to a cluttered code-base. To overcome this problem (and many others), powerful frameworks such as AngularJS and Ember are available. AngularJS requires JavaScript knowledge but provides powerful means to control the HTML-DOM directly and create a clean codebase for large projects by adding MVC capabilities directly into web-applications. It has become very mature in the last years and a whole JS world is established today providing all tools and means we are used to in professional enterprise software development: grunt (a build tool), bower and requireJS (dependency management), npm (package and tools management), WebStorm (IDE) ,…

Lessons-learned:

1) Know your team! Try to choose something that fits the available knowledge. But also don't underestimate the attractiveness of modern web-ui's. Many developers are happy to learn new technologies for developing nice-looking, responsive and mobile-fist web-apps.
2) Analyze your projects complexity and avoid writing complex grid-editing components in Angular if you can do the same in a 5-minutes Apex app!
3) Plan the costs of learning something new versus creating workarounds to make components you need available in the web-ui technology you have chosen.
4) Be professional! Develop your web-code just like the Java backend. Have a clear component/package structure, use dependency management, write unit tests, etc!

Choices and pitfalls in backend development

If using JaveEE to implement the backend, a default application server to use is probably given by your company. However, still many choices remain. Speaking of interfaces, important decisions are how you want to communicate with the frontend and the database backend. Frontend communication is usually done by thinking of the provided methods as services or resources and choosing SOAP or

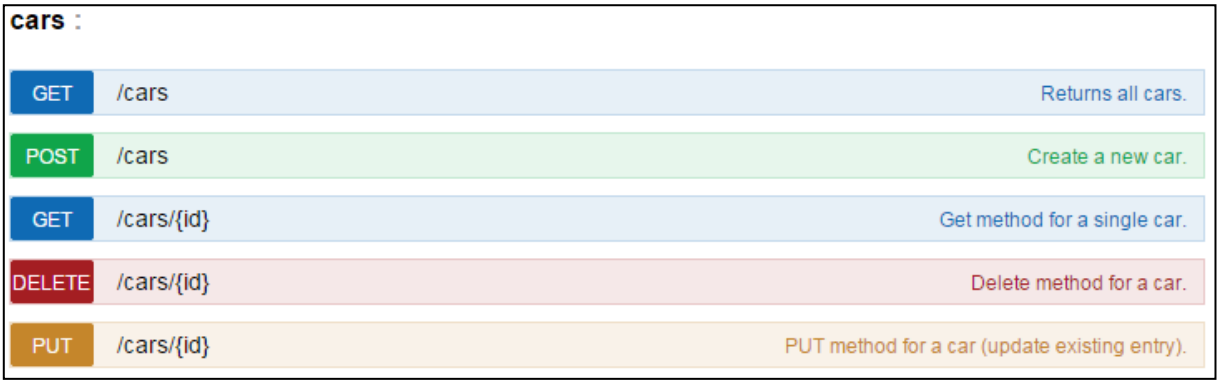

Figure 1: RESTful services API design example. „cars" is a resource, the colored boxes below are operations on this resource. A GET on the URL retrieves all elements, a POST creates a new element. If a specific ID is appended to the URL, an item is either retrieved (if an HTTP-GET is performed), removed (HTTP-DELETE) or updated (HTTP-PUT operation).

REST as the interface. In principal, REST describes the architectural model of applying CRUD (create, read, update and delete) operations on named resources and SOAP focuses on providing named operations. Today, the stateless REST services tend to supersede the stateful SOAP services. There are some rare occasions (especially in finance) in which you cannot replace SOAP by REST. However in most cases, REST services that use standard HTTP are not only easy to implement but also easy to test and consume. From an architectural point-of-view it is important to decouple the provided API from the frontend pages during planning. Do not try to provide methods for the operations you have on a certain page in your frontend. Rather think of the resources you have in your application (users/genes/parameters) and provide CRUD methods for them. An example of a good REST service architecture is shown in Figure 1.

Database communication is often handled by direct SQL (in small applications) or Hibernate/JPA in more complex applications. Especially the latter one is very common for reasons that we don't want to discuss here. Instead, we would like to suggest another approach: Using a separate PL/SQL API that is consumed by the Java backend. Considering the power of where/merge statements and the full Oracle-SQL capabilities, only parts of it can be covered with JPA. In fact, most projects using JPA often keep the database "dumb" and only use select/update/insert statements. They barely use the full power of the available Oracle technology. Some tasks might be a lot easier in plain (PL/)SQL with the full power available. Instead of JPA annotations, classes can be modeled in the database as Oracle types and tables of those. PL/SQL methods can directly be accessed from the Java Backend and the Java class (that corresponds to the Oracle type and implements the `ORAData` interface) can be passed directly to the method. This might ease, enhance and speedup the implementation in cases, in which complex inter-dependencies between tables or PL/SQL dependencies are needed.

Continuous integration (CI) with Java backends is also a common and highly recommended setup. Most systems like Jenkins or Bamboo can be extended to run custom operations after a build succeeded. This can be used, for example, to add a mechanism that automatically copies web-applications to the auto-deployment folder of your application server. As a result, a dedicated instance of the application can be configured that updates itself after developers make a commit and unit tests succeed. Speaking of unit tests, it is clear that these should be mandatory for all business logic. An existing CI system for the Java Backend is also a great opportunity to include tests for the Web- and PL/SQL code of your project, if no dedicated test environments for those are available. This way, you can directly test the PL/SQL API from the Java Backend unit tests. There are also good arguments against testing the real database and rather mocking the database interface. Most importantly, runtime of tests, database usage, different states (there might be unexpected content in database tables), simultaneous tests on the same database instance, etc. However, in a real-world usage of the application, simultaneous method calls are expected to work and after all, testing the PL/SQL code with the JUnit tests is better than not testing it at all. Just keep in mind that JUnit test failures might

occur not because of invalid Java code, but of somebody just changing the PL/SQL code behind it. Therefore, good error handling and meaningful error messages are a “must”.

Lessons-learned:

1) No matter what API is provided to the frontend – a clear API design is very important! Using tools for automatically documenting your API (like *Swagger* or *enunciate*) helps dedicated frontend developers to know about the API and to test it directly (without the need of backend developers to explicitly document the methods and their expected JSON objects).
2) Instead of creating a “dumb” database that purely stores objects in tables, use the provided Oracle features! If it comes down to complex inserts of sophisticated objects, think of a dedicated PL/SQL API instead of JPA in order to harness the full power of your database.
3) Organize and plan your code well! Refactor it from time to time or if the codebase becomes large and maintain a clear architecture following common rules within your codebase.
4) Go from the start with the final application server and test your code on the productive setup. It takes time, for example, switching from Tomcat to JBoss or Jersey-based REST services to a RESTEasy implementation.
5) Automated unit tests on a CI system are a must. Keep in mind to also test referenced code, such as PL/SQL implementations.

Kontaktadresse:
Dr. Clemens Wrzodek
pRED Informatics – Roche Innovation Center Penzberg
Roche Diagnostics GmbH
Nonnenwald 2
D-82377 Penzberg

Telefon: +49 (0) 8856 60 10587
E-Mail clemens.wrzodek@roche.com
Internet: www.roche.com