Chapter 1
Evolution of Web Systems Architectures: A Roadmap

Raoni Kulesza, Marcelo Fernandes de Sousa, Matheus Lima Moura de Araújo, Claudiomar Pereira de Araújo, and Aguinaldo Macedo Filho

1.1 Introduction

Web systems have become popular because of the Web browsers ubiquity. This characteristic allows us to conveniently install and maintain software systems on a server without changing client-side software, even if it is accessed by millions of browsers [15]. Currently, Web systems are used for all kinds of applications, such as e-commerce, audiovisual content access, email, social networks, searches, corporate portals, etc. [13].

Web systems can be considered a kind of client–server architecture model. In this scenario, the Web browser represents the client that interprets HTML, CSS, and JavaScript code. Besides, it communicates with the server using a URL and the HTTP protocol [7]. In the beginning, each Web page was delivered to the browsers as static documents and the server's responsibility was only to receive requests for locating and sending files. However, servers can now generate a dynamic page for each request by running software, accessing the database, or integrating with other systems. In addition, a Web page can also execute code on the client-side. These characteristics led to the creation of different software development platforms

R. Kulesza (✉) · M. L. M. de Araújo · C. P. de Araújo
Federal University of Paraiba (UFPB), João Pessoa, Brazil
e-mail: raoni@lavid.ufpb.br; matheus.lima@lavid.ufpb.br; claudiomar.araujo@lavid.ufpb.br

M. F. de Sousa
Institute of Higher Education of Paraiba (IESP), Cabedelo, Brazil
e-mail: marcelo@iesp.edu.br

A. M. Filho
Audit Office of Paraiba (TCE/PB), João Pessoa, Brazil
e-mail: amfilho@tce.pb.gov.br

V. Roesler et al. (eds.), *Special Topics in Multimedia, IoT and Web Technologies*,
https://doi.org/10.1007/978-3-030-35102-1_1

(languages, libraries, APIs, frameworks), both server-side and client-side [25, 26]. Such solutions are mainly written using Java, C#, Python, Ruby, or JavaScript, and there are hundreds of options [26].

Another important issue is that several Web systems have quickly become very important and have gained worldwide access. For example, Facebook has one billion hits every day and Netflix has 81.5 million customers in 80 countries [13]. These kind of systems need to meet increasingly demanding requirements, such as high availability and performance, scalability, security, multiple failure points, disaster recovery, transaction support, and integration with other systems [23]. Consequently, the client–server architecture has evolved in this software category and there are several models currently presented as a solution [5].

This chapter aims to study the main options of web-based software platforms, both on the client-side (React JS, Angular JS, and Vue JS) and the server-side (Spring and Node.js). In addition, we present the history and evolution of Web System's architectural models, such as 3 layers, n layers, RESTFul [30], and microservices [3]. Finally, we present solutions developed at the Paraiba Audit Office (TCE/PB) in partnership with the Digital Video Applications Laboratory (LAViD) of the Federal University of Paraiba (UFPB) in order to illustrate the practical use of technologies and architectural models in a real project. The main contribution of this work is the dissemination of the history of Web systems and the understanding of the technologies and architectures used today, as well as the trends for the future.

1.2 Fundamentals of Web Systems

1.2.1 History and Evolution of the Web

The Web—also known as the WWW or World Wide Web—was created by Tim Berners-Lee in the early 1990s and can be understood as a distributed and weakly coupled system for document sharing. Actually, Tim originally conceived the Web as a collaborative space where people could communicate through shared information [2]. However, the emergence of new technologies, such as cloud computing [24] mashups [31], among others, has boosted Web development. Thus, what was once a distributed system of interlinked documents became a platform for open, interactive, and distributed applications and services [21]. In order to understand the evolution of the Web, [1] proposed a taxonomy that was adapted by [4] that divides the story into three waves: (1) read only, (2) read/write Web, and (3) programmable Web. As we can see in Fig. 1.1, the so-called waves are not divided by time necessarily, but by the appearance of new functionalities and, in this way, they can overlap and coexist in certain periods.

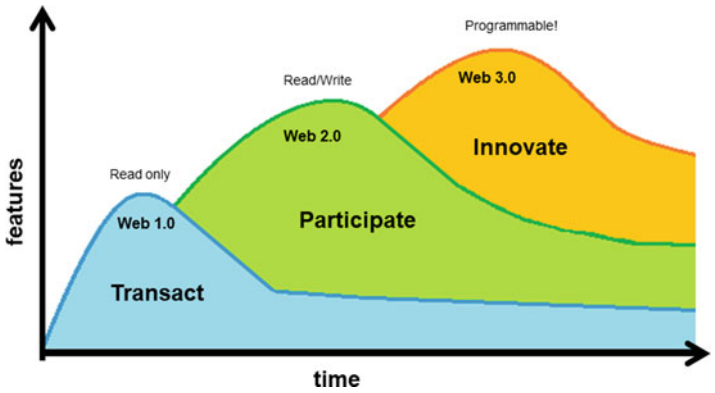

Fig. 1.1 Web evolution: adapted from [4]

The first wave of the Web (read only Web) is called Web 1.0 and has applications capable of providing information in a single direction, being limited in terms of communication and interaction between users. Therefore, applications that allow the realization of transactions of goods and knowledge such as search engines and e-commerce services belong to this first wave. The second wave of the Web (the read/write Web) is called Web 2.0 and has as its main characteristic the interaction in communities through participation, collaboration, and co-creation. In this way, social networks, blogs, etc., are representatives of this second wave. Finally, the third wave of the Web (programmable Web) is called Web 3.0 and has the feature of allowing anyone to create a new application or service from a web-supplied infrastructure. This wave is driven by the advent of cloud computing which allows the Web to take on the role of a platform for an ecosystem of people, applications, services, and even objects (Internet of Things—IoT).

1.2.2 URL and HTTP

To better comprehend the modern Web systems, it is necessary to understand the following fundamental concepts: resources and their representations; URIs; and actions (verbs). In the Web context, resources are data and information, such as documents, videos, or any device that can be accessed or manipulated through Web-based systems. Many real-world resources can be represented on the Web, requiring only the proper information abstraction to do so. This strategy makes the Web a heterogeneous and accessible platform, since practically anything can be represented as a resource and made available on the Web [30]. To identify, access, and manipulate resources published on the Internet, the Web provides the Uniform Resource Identifier (URI), which establishes a way to identify resources through a one-to-many relationship. This means that a URI identifies only one resource,

but one resource can be identified by many URIs. For example, a resource such as a Playlist can be represented by the HTML markup language and interpreted by Web browsers. In a similar way, the Playlist can also be represented in XML/JSON format, which is usually used by other systems and machines. Figure 1.2 exemplifies the representation of a resource with several URIs and representations. The Uniform Resource Locator (URL) is a URI that identifies the mechanism by which a resource can be accessed. For example, HTTP [9] (HyperText Transfer Protocol) URIs are examples of URLs. HTTP is an application layer protocol of the TCP/IP stack model used for data transfer over the Internet. It is through this protocol that resources can be manipulated. To do so, there are actions provided by HTTP. The original HTTP specification provides a series of request methods responsible for indicating the action to be performed on the representation of a given resource. These methods are also known as HTTP verbs. The HTTP verbs used for interaction with Web resources are GET: is used to request a representation of a specific resource and should only return data; HEAD: similar to the GET method, however, it does not have a body with the feature; POST: is used to submit an entity to a specific resource, possibly causing a change in resource state, or requesting server-side changes; PUT: requests data load replaces all current representations of your resource; DELETE: removes a specific resource; CONNECT: establishes a tunnel for connection to the server from the target resource; OPTIONS: describes communication options with the target resource; TRACE: runs a loopback call as test during the connection path to the target resource; PATCH: applies partial modifications to a specific feature.

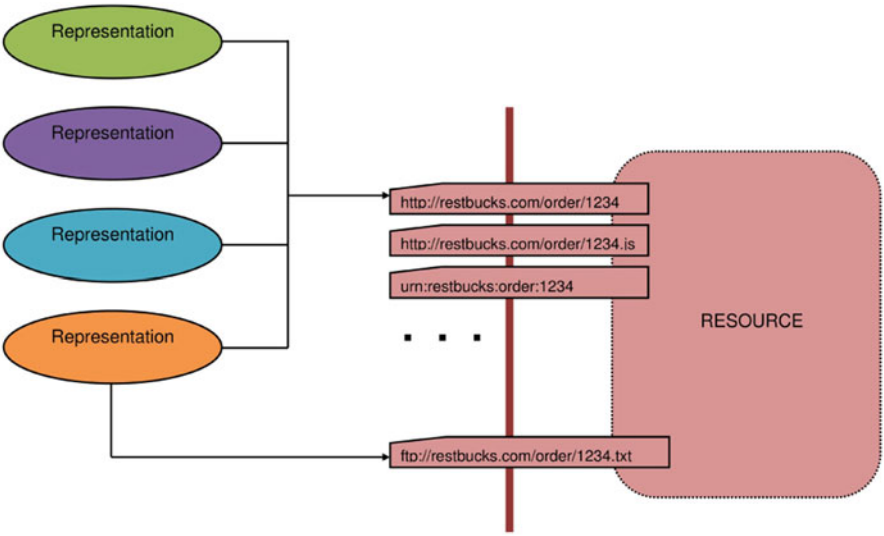

Fig. 1.2 Web principles [30]

1.3 Client Technologies for Web Systems Development

The success of the Web increased its access, as well as the complexity of the available content, that evolved from static pages to applications with the capacity to behave in a similar way to desktop applications. In this context, in 1995, Netscape Communications introduced JavaScript, a client-side scripting language that allows programmers to enhance user interface and interactivity with dynamic elements. A few years later, in 2005, Jesse James Garrett proposed an approach to Web application development called AJAX (Asynchronous JavaScript + XML). Until then, user interactions (client-side) in a Web application submitted HTTP requests to a server that returned a new HTML page. Garrett's proposal brought a significant change to this traditional method, adding a layer responsible for requesting data from the server and performing all processing without the need to update the entire HTML document structure, thus making the communication between client and server asynchronous [14] HTML pages have become more user friendly with the advent of AJAX, since it allows to update parts of a web page without reloading the whole page. However, the JavaScript language had to face the competition between Web browsers that developed specific solutions for their products that were often incompatible with rival Web browsers. This scenario motivated the JavaScript community to implement libraries and frameworks, such as jQuery, to mitigate this problem, offering uniform behavior and productivity. Following the success of AJAX, the consolidation of both HTML5 and the tools used to improve the user interface development, the concept of Single Page Application (SPA) has emerged. This is a type of application that loads a single HTML page along with its JavaScript and CSS resources. After that, the browser becomes responsible for dynamically rewriting the current page instead of loading whole new pages from a server, minimizing client–server traffic. Thus, the browser supports more programming logic, being able to perform tasks such as HTML rendering, validation, UI changes, and so on [22]. JavaScript has grown a lot over the years and has an active community. Nowadays, developers have several modern alternatives for developing user interface with JavaScript based frameworks. For example: AngularJS, Ember, ReactJS, VueJS. In the Sect. 1.3.2, we will introduce you to ReactJS.

1.3.1 Single Page Application

The single page application, or SPA, is based on the idea that the entire application runs as a single web page designed to provide a user experience similar to that of a desktop application. The presentation layer that was previously handled by the server was factored to be managed from the browser. As a result, single page applications are able to update parts of an interface without necessarily sending or receiving a full page request from the server, thus improving performance and user experience in most cases [28]. In a SPA, browser updates are not required until the

initial page load all the tools needed to create and display previews are downloaded and ready to use. If a new view is needed, It will be generated locally in the browser and dynamically attached to the DOM (Document Object Model) via JavaScript.

In a SPA, we can use different approaches to rendering server data. An example of this is partial server-side rendering, where we can combine HTML snippets with server response data. One of the most used approaches is to let the client render and only data is sent and received during business transactions. Commonly, one of the data-exchange formats for this type of data is JavaScript Object Notation (JSON), but other types of formats can be used, such as Extensible Markup Language (XML).

1.3.2 ReactJS

Throughout the history of the Web, several JavaScript libraries have been developed to address the problems of dealing with complex user interfaces. However, these libraries still maintained the classic separation of responsibilities that divides style (CSS), data, structure (HTML), and dynamic interactions (JavaScript). ReactJS is a JavaScript library for the development of user interfaces created and maintained by Facebook [8]. Unlike other approaches, ReactJS follows a component-based development approach. Thus, instead of defining a single model for the interfaces, they are divided into small reusable components, so the principle is to reduce complexity through component separation [19]. Therefore, ReacJS facilitates reuse, in addition to other benefits such as maintenance and distributed development, and easily promote integration with the development process. It is worth noting that the development of componentized user interfaces (UIs) is not a new approach, however, React was the first to do so from pure JavaScript without the use of models. In React, you can focus on your view layer before introducing more aspects to your application. React is not a complete JavaScript front-end framework and does not establish a specific way to develop modeling, style, or routing of data. React acts as the "V" of the MVC architecture model. Therefore, developers use React along with a routing or modeling library. The developer is free to choose which libraries to use, but there is a React Stack widely adopted to develop a complete front-end application [19]. This stack consists of data and routing libraries designed to be used specifically with React. For example, the RefluxJS, Redux, Meteor, Flux are used for the data model. The React Router is recommended for routing library. Finally, for user interface styling the React component collection that consume the Bootstrap Twitter library, the React-Bootstrap can be used.

To facilitate development in ReactJS, JavaScript Syntax eXtension (JSX) was developed. JSX is a syntax extension for writing JavaScript as if it were XML. It does not run in the browser, but is used as the source code for compilation. It is transpiled in regular JavaScript. It is optional but is recommended by Facebook for React application development. Although it sounds like a model language, JSX has the same power as JavaScript and produces React elements.

In a React application you need to think about using the component-based architecture, which allows you to reuse code by separating functionality into loosely coupled parts. By adopting this strategy, code becomes scalable, readable, reusable, and simple to maintain. This abstraction allows reuse of user interfaces in large and complex applications as well as in different projects. Standard HTML tags (div, input, p, h1, etc.) can be used to compose React component classes as well as other components. This allows for flexibility in creating robust and potentially reusable components. Theoretically, the components in React are like JavaScript functions. It is possible to provide data entries called "props," and they return React elements that describe what should be displayed on the screen. You can define a component in several ways, the simplest one being through a JavaScript function.

In addition to ReactJS, other client-side frameworks have been created those have also become famous among developers, namely Angular and Vue.js. Angular is popular web framework, from Google, built and based on Typescript. In addition, Angular uses well-known concept for components, DOM and virtual models. Code for templates can also be placed in a separate HTML template file. Like ReactJS, Vue.js is also based on Virtual DOM, but its implementation is different from ReactJS. Its implementation is optimized for efficiency, which means it only updates those DOM elements if really need it. Instead of using JSX, Vue uses its templates. These feature easy-to-use and readable syntax for creating your UI.

1.3.2.1 Single Page Application and ReactJS

React makes it possible to develop a SPA, although it also allows other alternatives. Code written in React can coexist with markup rendered on the server by something like PHP or other client-side libraries. For example, assuming SPA uses an MVC architecture, the application navigator acts as the "C" of the MVC architectural standard, and determines which data to fetch and which model to use. It also performs requests for data collection and populates the views from the data obtained to render the user interface. The UI sends actions back to the SPA, such as mouse events, keyboard events, etc. [19].

1.3.2.2 Virtual DOM

A fundamental concept that makes ReactJS applications different is the Virtual Document Object Model (VDOM). This is a programming concept in which a virtual representation of the user interface in pure JavaScript is created, kept in memory and synchronized with the actual DOM (Document Object Model) by a library such as ReactDOM [8]. Therefore, the application interacts with VDOM instead of the DOM. The main reason for this is to avoid performance issues, because if DOM updates its structure directly, several unnecessary updates would be executed, causing performance issues, especially in cases where the user interface is complex [19]. Thus, with each change in the VDOM, an algorithm first calculates

the difference between the VDOM and the real DOM and, from this analysis, the library is able to identify the change in the rendering, updating only the change in the real DOM [6].

1.4 Architectural Patterns for Implementing SPA

It is essential in a SPA to keep code segregated based on its functionality. Taking this approach, application code is easier to design, develop, and maintain if segmented based on the type of responsibility each layer has. The SPA can be broken into multiple application layers, both server and client-side. Architectural patterns have emerged to help developers build robust and scalable applications [28]. This section details some of the most successful patterns in client-side approaches to building SPAs, namely: MVC, MVP, MVVM, Flux, Redux.

MVC is one of the oldest standards. This pattern is based on the idea of separating the application into three layers called data, logic, and presentation. The MVC pattern includes the Model, the View, and a Controller. The model contains data, business logic, and validation logic. The model notifies the view of state changes but never cares about how data is presented [28]. The controller is responsible for user interactions and sending commands to the model to update its state. The view is aware of the model in this pattern and is updated when changes are observed.

In MVP, the role of the controller is replaced with a Presenter. The MVP is a variation of MVC. The purpose of this pattern was to increase dissociation between the model and the other two components of MVC. The view delegates actions to the presenter. The Presenter has direct access to the model for any necessary changes and calls methods on the view to notify it to update itself [28]. So, the presenter is responsible for mediate the actions between the model and the view.

The MVVM is based on MVC and MVP, which tries to make UI development even more isolated from behavior and business logic in an application. The MVVM pattern includes Model, View, and ViewModel. As in MVP, the view itself is the point of entry. The ViewModel is a model or representation of the view in code, in addition to being the middleman between the model and the view [28]. It changes Model information into View information, passing commands from View to Model.

Flux is an architectural pattern that was developed as an alternative to traditional MVC architectures or their derivatives. Basically, it is an architecture designed to avoid the concept of multidirectional data flow and linking, which is common in typical MVC structures. The components that make up this architectural pattern are Actions, Dispatcher, Store, and View [8]. The Action is a simple object containing the new data and an action ID type property that is dispatched to the Store. Dispatcher is responsible for managing the entire data flow in a Flux system. It is important to note that it is not the same as the MVC Controller, as Dispatcher does not usually have much logic inside it. The Store contains all the logic and state of an application. Dispatcher, Store, and Views are independent nodes with distinct inputs and outputs. The View observes state changes emitted by the store.

Redux comes up with the idea of enhancing the original design of the Flux pattern by creating a single global singleton Store that stores state for each existing View in the application. Redux, like Flux, emphasizes the importance of unidirectional data flow. Redux is based on three principles. The first is that the entire state of the application is contained in a centralized repository, a single store, acting as the only source of truth of the system, which differs from the Flux model, where it is possible to have different stores, each responsible for its own logical domain. The second principle of Redux is that the state of the application is immutable, that is, the object representing the state should not be directly modified [20]. The third principle says that all functions that calculate a new state, in this case Reducer functions, must be pure functions. Pure functions are functions that have no side effects and are deterministic, that is, for a given set of inputs, the output will be the same. In Redux, this state is modified by Reducers that modify bits and pieces of the global app state.

1.5 Web Systems Architecture

Web system architectures have evolved considerably since the beginning of the Internet. At first, the systems were developed using the CGI (Common Gateway Interface) architecture. CGI gave a lot of power to the servers, since it started to offer the ability to execute code scripts—usually Perl—when processing HTTP requests, making Web systems able to process requests in a more dynamic way [18]. Another problem at the beginning of the Web was the difficulty in developing the interface code and the business logic code of the Web applications in a separate way. In order to do so, the template systems emerged, which allowed executable codes of a programming language to be injected directly into the files responsible for presenting the system. Thus, the 2 layers (presentation and logic) were better divided [10]. After that, a number of architectures emerged, such as MVC's "Model 2" architecture, which later became one of the main Web systems model [29] and pushed technologies such as Struts, Tapestry, and Java Server Pages (JSF). Also at that time, frameworks were developed to facilitate the mapping between object-oriented models and relational models, such as Hibernate, which served as the basis for 3-tier architecture (presentation, business logic, and data) [10]. Figure 1.3 presents a typical 3-tier architecture for enterprise applications with a view layer, a controller layer, and a model layer. As we can see, a request is realized by the user through the browser defining which view of an application will be presented. So, the view triggers the controller that can retrieve the information directly in the model or call business services in order to aggregate data from different sources. Finally, the model classes provide the mapping onto the data storage and are passed back up through the layers [11].

Due to the growing use of systems in a corporate environment with global access, it was necessary to divide the processing from 3 to n layers [12]. So, distributed execution platforms such as Java Enterprise Edition (JEE), .NET, and Spring emerged.

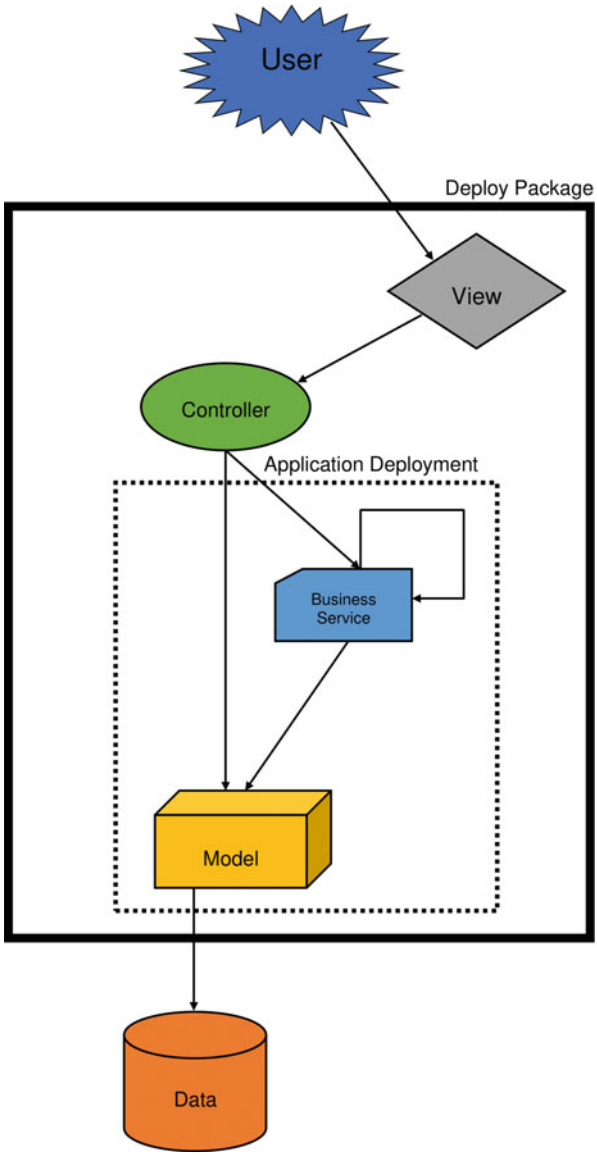

Fig. 1.3 Typical enterprise application architecture: adapted from [11]

Communication protocols (SOAP, REST, etc.) also appear, allowing systems to communicate regardless of the programming language, facilitating the integration of heterogeneous and legacy systems. As a result, developers were no longer just developing applications that served content to browsers; but rather complex systems that involved multiple layers of internal and external communication (with

other systems) [17]. Besides, real enterprise applications usually diverged from the clean architecture presented in Fig. 1.3 that presents clear boundaries between functionality within a layer. This situation happens because of a number of reasons, such as deadlines pushing the development team, changes in development team over time, difference of architecture preferences by the developers, etc. In this context, the boundaries between functionalities become blurred, resulting in components in each layer no longer having a well-defined purpose [11]. From then on the systems grew a lot, and the number of users increased considerably, causing these systems to become too large, turning them into giant monolithic systems [23].

A monolith, according to [11], is an application that has all its components contained within a single deployable, usually does not respect boundaries between functionalities and has a release cadence of 3–18 months. It is also a common characteristic multiple deploy packages that are part of a single deployment. These systems have many scalability and performance issues when many users use it, but it is worth mentioning that the concept is more related to the coupling of dependencies between components, leading to a problem when updating a single component is necessary. The ideal scenario is to perform this task without needing to cascade updates across many components, which allows a faster release cadence [11]. The solution was found in less monolithic and more distributed architectures, such as service-oriented architecture (SOA), using the concept of microservices and polyglot persistence [27]. These models have a better distribution of each system service, making the request load better distributed, greatly improving scalability requirements such as load balancing and high reliability [23].

Figure 1.4 demonstrates a typical microservices architecture for enterprise applications. We can understand a microservice as a single deployment executing within a single process, isolated from other deployments and processes, responsible to do one thing well. In other words, a microservice accomplishes a specific business functionality, which is a logical way to separate the domain models of an enterprise. A microservices architecture becomes useful when containing many microservices loosely coupled communicating with each other and working together [11].

As mentioned, major breakthroughs have also been achieved at the client-side presentation layer through frameworks that enable Web systems to have performance and usability comparable to traditional desktop systems [28]. These frameworks use SPA architectures [28], updating only what is needed through the use of newer versions of JavaScript (e.g., ECMAScript versions 5 and 6) and AJAX server communications [28]. This model removes the responsibility for generating the view of servers, making systems lighter and faster [28]. In this context, in [26] we can find that there are currently numerous options for development platforms (languages, APIS, libraries, frameworks, etc.) for systems. The next section describes a case study that demonstrates a set of state-of-the-art options regarding the use of technologies (client and server) and the application of modern architecture concepts to Web systems.

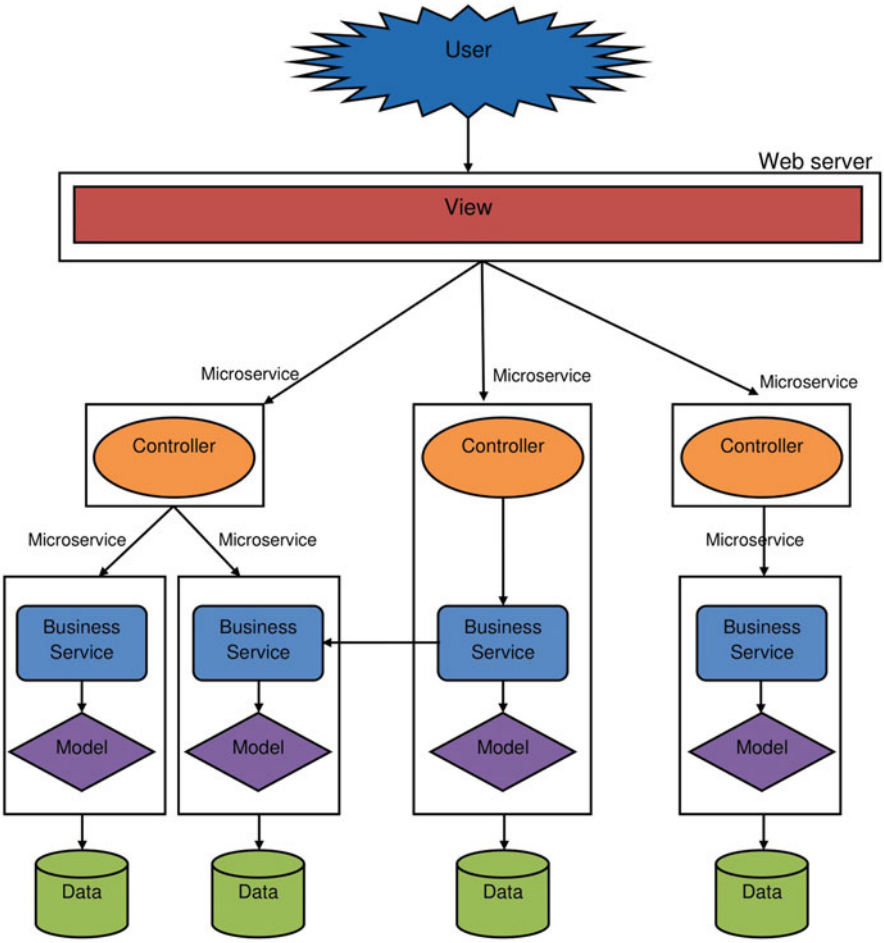

Fig. 1.4 Enterprise microservices: adapted from [11]

1.6 Case Study: *Você Digital*

Você Digital[1] is a research and development project conducted by the Paraiba Audit Office (TCE/PB) in partnership with the Digital Video Applications Laboratory (LAViD) of the Federal University of Paraiba (UFPB) for modeling and development of a collaborative computing platform for electronic government (e-government). The main objective is to improve political engagement by automating popular compliments and complaints, which enables a better interaction and communication between society and public administration. In addition, the platform

[1] Available at: http://controlesocial.tce.pb.gov.br/.

aims to exploit the collective intelligence present in the networks, promoting citizen participation that can help reduce TCE/PB operating costs, increase transparency, reliability, and efficiency of services promoted by the virtual and democratic communication channel of society's demands through the proposal tool. In addition, as a digital public management tool, the idea is to evaluate part of public services, as well as to encourage popular participation in the decision-making process of public auditors and managers. In order to evaluate the platform, a mobile application has been developed that will use new methods capable of increasing citizen involvement in the context of problem diagnosis in different areas of public service (e.g., education, health, and safety).

1.6.1 Vsocê Digital *Architectural Project*

Figure 1.5 presents the high-level architectural design of the *Você Digital* platform with its subsystems highlighted in blue. Possible TCE/PB internal systems are highlighted in orange and external systems are represented in the upper corner of

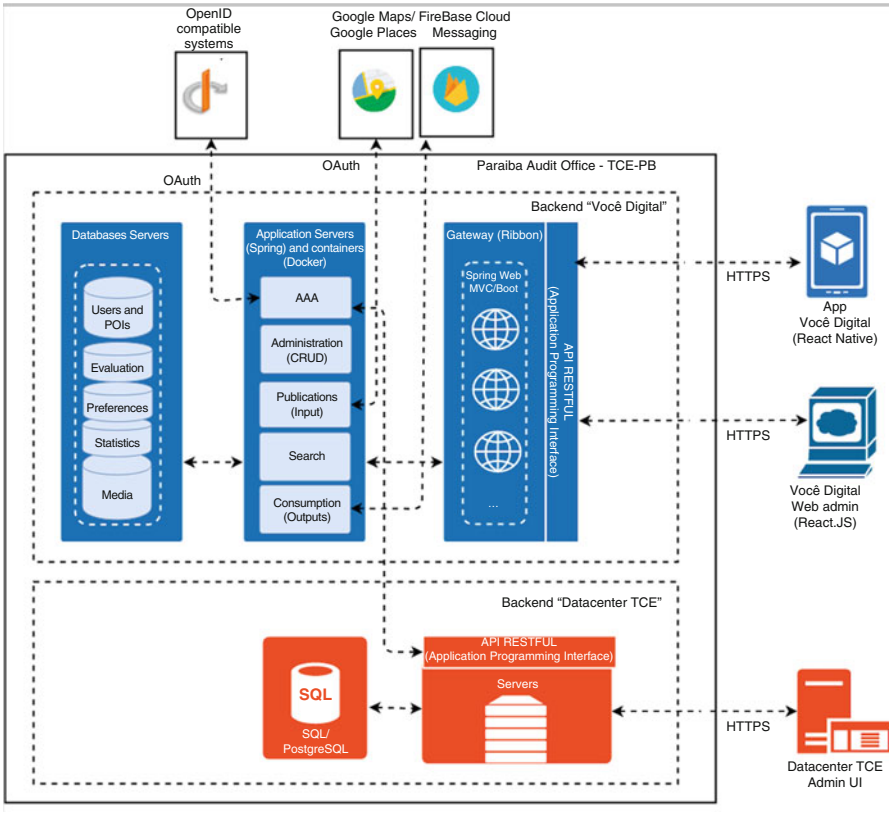

Fig. 1.5 *Você Digital* architectural design overview

the illustration ("Systems with OpenID," "Google Maps and Google Places"). Both parts of the *Você Digital* system and TCE-PB internal systems will use the data center and virtualization infrastructure currently available on TCE/PB. The *Você Digital* system consists of two large subsystems: (1) 2 (two) client software systems (front end) and (2) 1 (one) back-end system. The first (1) set has a mobile app built with React Native technology (see Fig. 1.5 *Você Digital* App) available for download from Apple and Google Web Stores. In addition, there is also a client software system (see Fig. 1.5, *Você Digital* Administration) that allows to manage the *Você Digital* system through tasks such as data management, registration management, user permissions, statistical reports, etc. This application is based on the Single Page Application (SPA) approach and React technology to enable it to run on any web browser (desktop or mobile device). The second set (2) is responsible for the processing of data registers available in the system, as well as inferring through this data information for users. This subsystem is divided into three parts: (I) **Controller**: Responsible for load balance, high availability, and secure access to data and information available to applications through a RESTful API; (II) **Application servers** (containers based on Docker platform and technologies for the development of microservices architectures) responsible for handling the integration and processing of internal and external data and partitioning of the functionalities available for client software and (III) **Database Systems**: responsible for storing data using polyglot persistence (this module uses SQL and/or NoSQL technologies). Communication between applications (1) and servers (2) is accomplished through HTTPS protocol and RESTFul APIs.

Regarding the integration requirements of the *Você Digital* system with external systems, APIs available on OpenID-compliant systems were used to allow external authentication (e.g., social networks) so no registration is required on the *Você Digital* system to access application services. Similarly, the Google Maps APIs were used to obtain geolocation information (Google Maps) and geo-localized points of interest registered by individuals and legal entities (Google Places). For the integration with the internal systems of the TCE/PB, a mapping was made of what data and/or information would be needed. Thus, the TCE/PB team offered a RESTFul API for communication between systems. Similarly, the *Você Digital* system offers APIS RESTFul for TCE /PB to access data. According to Fig. 1.5, the application server subsystem has been organized into the following components: **AAA** (Authentication, Authorization, and Accounting): these are the procedures related to authentication, authorization, and auditing. As is well known, authentication verifies the identity of users, authorization handles permissions, that is, it ensures that an authenticated user only has access to the resources authorized for their profile and, finally, the audit is related to the action of collecting data on user behavior in relation to the system. It is noteworthy that this module communicates with external authentication services and is responsible for managing the sections; **Administration** (CRUD): is the module that manages all entities of the class model, being responsible for performing the four basic operations of creation, query, update and destruction in database; **Publications** (Inputs): this module is the main source of data input of the *Você Digital* system and it is responsible for

receiving all user-generated information such as ratings, comments, video and photo recordings. In addition, it also has the responsibility to handle data security. Due to the nature of the system, it is necessary to apply text filters to comments to identify inappropriate posts, as well as to sanitize data to prevent HTML injection attacks. Another competency for this module is to provide application authentication mechanisms to prevent fraud through artificial intelligence programs (robots) that can be used for information manipulation; **Search**: handles low granularity searches, such as miscellaneous database queries; and finally, **Consumption** (Outputs): this module uses the Search module to perform Data Analytics in order to generate statistical data, data transformation, graphs, reports, and other analyzes. Continuing to detail the architecture, the database server subsystem was organized in the following bases: **Users and POI**: this base stores the user registration information and their points of interest; **Evaluation**: this database stores information related to the history of evaluations performed by users; **Preferences** (profile): this database stores dynamic information related to users, such as: IP, Latitude and Longitude, among others; **Statistics**: this database stores persistent statistics that can be used by the Consumption module of the application server subsystem; and lastly, **Media**: which is a base that stores all user-generated media, such as text, images, videos, and audios. From a technology standpoint, the Spring ecosystem was adopted for the implementation of the Application Server subsystem. The Spring framework is a tool used to increase productivity in writing enterprise applications by exploring concepts such as dependency injection and inversion of control. In addition to Spring technology in the development of business logic and data access on the server, the Spring Cloud Suite Solutions also has been adopted on the Controller subsystem, which provides functionality for configuration, routing, load distribution, and high availability for implemented services.

1.6.2 Você Digital *Frontend Architectural Project*

The *Você Digital* frontend application architecture, as shown in Fig. 1.6, follows the Redux architectural pattern. In Fig. 1.6 the Views are composed of the components of React. Actions will be triggered from interactions in Views by the user, following to Reducers, in case of synchronous requests. Asynchronous requests will be handled in the Redux Thunk Middleware,[2] which will query the APIs used in the system, in this case the *Você Digital* API, the Google Maps API, and the Google Places API. After the API result returns, the result goes to the Reducers along with the current state of the application. The Reducer responsible for the dispatched Action will update and return the new state of the application, updating the Listener Views.

[2] Available at: https://github.com/reduxjs/redux-thunk.

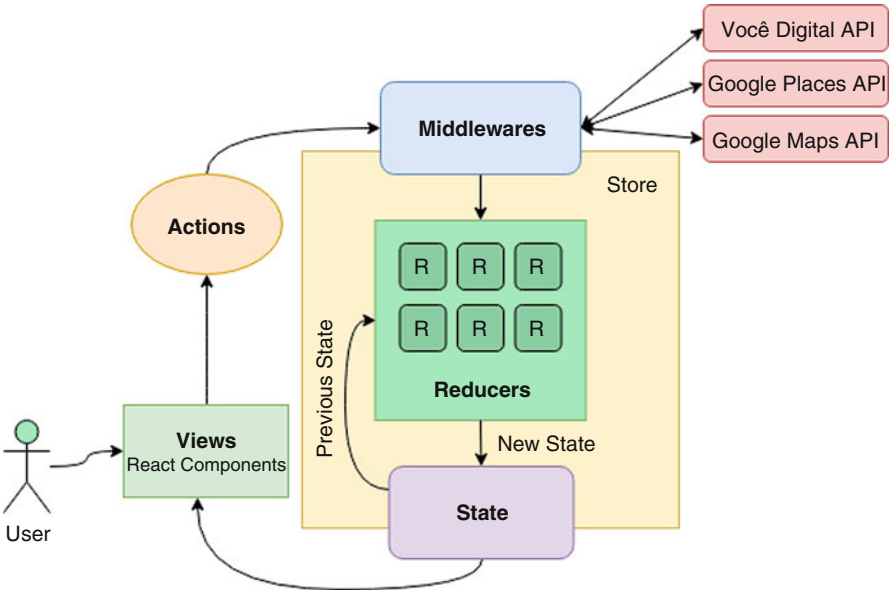

Fig. 1.6 *Você Digital* frontend architectural project

The main architectural difference between the mobile application architecture (React Native) and the web application architecture (ReactJS) is how to render Views. While React.js uses Virtual DOM React Native uses native iOS or Android APIs.

ReactJS uses the virtual DOM. DOM building takes time because DOM trees are big today. But ReactJS can perform this procedure faster using a virtual DOM. ReactJS then uses an abstract copy of the Document Object Model and distributes the changes into one component without influencing the rest of the UI.

React Native uses native APIs to render parts of the UI that can be reused on iOS and Android platforms. So what it really does is use Java APIs to render Android components and Objective-C APIs to write iOS components. JavaScript is not a language that runs natively on the mobile device, it is executed on an interpreter known as JavaScript Core Engine and communicates with native APIs via a JavaScript bridge [16]. It then uses JavaScript to compose whatever remains of the code, individualizing the application for each platform. This gives React Native mobile applications maximum component reuse and code coding capability.

1.7 Final Remarks

This chapter was a brief roadmap of Web technologies related both client and server-side software development platforms. It presented the history of architectural models evolution of Web systems. It also presented a case study through the

solutions developed at the Paraiba Audit Office (TCE/PB) in partnership with the Digital Video Applications Laboratory (LAViD) of the Federal University of Paraiba (UFPB). The Você Digital solution has adopted the technologies and architectural models discussed in a real project. The main contribution of this work is to disseminate the history of Web systems and to elucidate the technologies and architectures used today and trends for the future.

References

1. Benioff, M.: Welcome to Web 3.0: Now Your Other Computer is a Data Center. TechCrunch. http://techcrunch.com/2008/08/01/welcome-to-web-30-now-your-other-computer-is-a-data-center-2/ (2008). Cited 16 Jul 2019
2. Berners-lee, T.: WWW: past, present, and future. Computer (1996) https://doi.org/10.1109/2.539724
3. Boner, J.: Reactive Microservices Architecture. Pearson Education, Sebastopol (2016)
4. Buregio, V.A.A.: Social machines: a unified paradigm to describe, design and implement emerging social systems. Doctor of Computer Science (PhD): Computer Science, Federal University of Pernambuco, Recife, Brasil (2014)
5. Burns, B.: Designing Distributed Systems: Patterns and Paradigms for Scalable. O'Reilly Media, Sebastopol (2018)
6. Chedeau, C.: React's diff algorithm. Performance Calendar. https://calendar.perfplanet.com/2013/diff/ (2013). Cited 16 Jul 2019
7. Deitel, P., Deitel, H., Deitel, A.: Internet e World Wide Web: How to Program. Pearson Education, Boston (2012)
8. Facebook (2018): React – A JavaScript library for building user interfaces. ReactJS. https://reactjs.org. (2019). Cited 23 Jul 2019
9. Fielding, R., Reschke, J.: Hypertext transfer protocol (http/1.1): semantics and content. Internet Engineering Task Force (IETF) (2014). https://tools.ietf.org/html/rfc7231. Cited 23 Jul 2019
10. Fields, D.K., Mark, A.: Web development with JSP. Manning, Shelter Island (2002)
11. Finnigan, K: Enterprise Java Microservices. Manning Publications, Shelter Island (2019)
12. Fowler, M.: Patterns of Enterprise Application Architecture. Addison Wesley, Boston (2002)
13. Fox, R., Hao, W.: Internet Infrastructure. Taylor & Francis Group, New York (2018)
14. Garrett, J.J.: Ajax: a new approach to web applications. semanticscholar. https://pdfs.semanticscholar.org/c440/ae765ff19ddd3deda24a92ac39cef9570f1e.pdf (2005). Cited 23 Jul 2019
15. Groef, W.: Client- and Server-Side Security Technologies for JavaScript Web Applications. Doctor of Engineering Science (PhD): Computer Science, Faculty of Engineering Science, Ku Leuven, Leuven (2016)
16. Hansson, N., Tomas, V.: Effects on Performance and Usability for Cross-Platform Application Development Using React Native (2016)
17. Holdener, T.: AJAX the Definitive Guide. 1st edn. O'Reilly Media, Sebastopol (2008)
18. Hunter, J., Crawford, W.: Java Servlet Programming, 2nd edn. O'Reilly Media, Sebastopol (2001)
19. Mardan, A.: React Quickly: Painless Web Apps with React, JSX, Redux, and GraphQL. Manning Publications Co., Shelter Island (2017)
20. Masiello, E., Jacob, F.: Mastering React Native. Packt Publishing Ltd., Birmingham (2017)
21. Maximilien, E.M., Ranabahu, A., Gomadam, K.: An online platform for web APIs and service mashups. IEEE Internet Comput. (2008) https://doi.org/10.1109/MIC.2008.92
22. Mikowski, M., Powell, J.: Single page web applications: JavaScript end-to-end. Manning Publications Co., Shelter Island (2013)

23. Newman, S.: Building Microservices. Pearson Education, Sebastopol (2015)
24. Patterson, D., Fox, A.: Engineering Long-Lasting Software: An Agile Approach Using SaaS and Cloud Computing. Strawberry Canyon LLC (2012)
25. Raible, M.: Comparing Hot JavaScript Frameworks: AngularJS, Ember.js and React.js. Raible Designs. https://raibledesigns.com/rd/page/publications (2015). Cited 16 Jul 2019
26. Raible, M.: Front End Development for Back End Developers. Raible Designs. https://raibledesigns.com/rd/page/publications (2017). Cited 16 Jul 2019
27. Sadalage, P.J., Fowler, M.: Nosql Distilled: A Brief Guide to the Emerging World of Polyglot Persistence, 1st edn, Addison Wesley, Boston (2013)
28. Scott, Jr. E.A.: SPA Design and Architecture Understanding Single-Page Web Applications, 1st edn. Manning Publications, Shelter Island (2016)
29. Sommerville, I.: Software Engineering, 10th edn. Pearson Education, London (2015)
30. Webber, J., Parastatidis S., Robinson I.: REST in Practice: Hypermedia and Systems Architecture. O'Reilly Media, Sebastopol (2010)
31. Yu, J., Benatallah, B., Casati, F., Daniel, F.: Understanding mashup development. IEEE Internet Comput. (2008). https://doi.org/10.1109/MIC.2008.114

Raoni Kulesza, DSc., is an adjunct professor at the Federal University of Paraiba/ Informatics Center (UFPB/CI) and researcher of Digital Video Applications Laboratory (LAVID) where he teaches and coordinates multimedia systems projects, including Web systems. Ph.D in Computer Science at Federal University of Pernambuco (UFPE), MSc in Electrical Engineering at USP and B.S. in Computer Science at Federal University of Campina Grande (UFCG). He has been working with Web systems development for 20 years in projects for e-commerce, multimedia content management, digital TV and video transmission management, social networks and intensive data processing systems integrated with mobile devices.

Marcelo Fernandes de Sousa, DSc., is Ph.D. in Computer Science at Federal University of Pernambuco (UFPE), MSc and B.S. in Computer Science at Federal University of Paraiba (UFPB). He is professor and coordinator of computer graduation courses at Institute of Higher Education of Paraiba (IESP). He is currently a researcher at the Digital Video Applications Laboratory (LAVID), having worked on the GIGA-VR, Ginga Middleware and RH-TVD CAPES projects. He is mainly interested in the following subjects: Software Engineering, Digital Television, Interactivity, Ubiquitous Computing, Multimedia, MulSeMedia and Web systems.

Matheus Lima Moura de Araújo, B.S., is a MSc. candidate and received a degree in Computer Science (2018) from UFPB. He is a researcher member at the LAVID. His research interests are in the development of the client-side, mainly web and mobile applications.

Claudiomar Pereira de Araújo, B.S., is a software developer with 2+ years of professional experience and bachelor's degree (2018) in Computer Science at UFPB. He was a researcher at LAVID, where contributed to multimedia and Web systems. Currently, he works at Indra Company providing solutions for Web systems.

Aguinaldo Macedo Filho, MSc., is Accounts Auditor and Special Technical Advisor of the Intelligence Division of Paraiba Audit Office (TCE-PB). He is B.S. in Computer Science at UFCG and MSc in Computer Networks at UFPE. He has experience in project management of Web systems developments for the Brazilian Credit Protection Service, ANATEL, Ministry of Justice and TCE-PB.