ArRESTed Development

Guidelines for Designing REST Frameworks

A key challenge in developing RESTful Web systems is the lack of software development frameworks that support REST principles. This article gives practical guidelines for designing frameworks for developing such systems. Derived from intuitive formal models, these guidelines enable a development process that improves separation of concerns and the modifiability of developed systems. The authors analyze several existing Web frameworks to determine how well they correspond to these guidelines.

Ivan Zuzak
University of Zagreb

Silvia Schreier
University of Hagen

The REST software architectural style[1] is a major contributor to the Web's success. REST describes how large-scale distributed hypermedia systems, such as the Web, should operate to maximize beneficial properties, including scalability, modifiability, performance, simplicity, and reliability. To maintain usability in the face of continuing growth and expansion into new domains, the Web must retain the benefits of this RESTful design. We can see such growth in the introduction of new media types and protocols, such as SPDY and the Constrained Application Protocol (CoAP); the proliferation of machine agents driving the Web of Things; and the growing diversity in devices connected to the Web. However, although REST principles have been known for more than a decade, developing systems that conform to them is difficult.

Software frameworks reduce development complexity by providing technology implementations and principled guidance for the development process. Although most existing Web development frameworks offer implementations of Web technologies, such as protocols and media types, they don't provide adequate guidance for incorporating REST principles. Without such guidance, engineers often break REST principles, leading to systems with diminished modifiability, scalability, and performance. Consequently, both engineers and researchers consider inadequate tool support a significant drawback to RESTful development.[2,3]

Here, we present guidelines for designing frameworks for developing RESTful systems. Two aspects of system development drive these guidelines. First, frameworks should provide a greater separation of concerns to increase reusability and modifiability, and to focus development efforts on domain expertise. Second, complex engineering disciplines should utilize theoretical foundations for practical guidance.[4] Thus, frameworks should use

 Published by the IEEE Computer Society 1089-7801/12/$31.00 © 2012 IEEE

Related Research and Practice in REST Framework Development

Existing research[1,2] indicates that RESTful development doesn't benefit from integrated development environments (IDEs) or require client code generation, but rather needs tools that give developers guidance on following REST principles. Recent literature presents practical patterns for developing and consuming RESTful Web services,[3,4] such as designing URIs and versioning Web services. Furthermore, several authors express requirements for testing the RESTfulness of Web services,[2,5–7] with a strong focus on supporting hypermedia-driven service development and consumption. Although not directly guiding framework implementation, these results recognize the uniform interface principle as the key issue of RESTful development and discuss the benefits and challenges of implementing hypermedia-based systems.

Issues with understanding REST, building machine-to-machine systems, and documenting applications have motivated research on formal models,[8,9] such as finite-state machines, for describing RESTful systems[10] and application-domain protocols.[11] These results indicate that framework developers can use formal models of RESTful systems as the foundation of RESTful frameworks if they leverage those models to provide suitable development abstractions that encapsulate REST principles.

Finally, some authors express requirements for RESTful frameworks[12] and implement frameworks for developing RESTful Web systems. Although some frameworks are maturing in their support for RESTful development, a growing need exists to define generic guidelines for directing framework and system design. We analyze several Web frameworks in the main text that provide noticeable support for implementing REST principles.

References

1. C. Pautasso, O. Zimmermann, and F. Leymann, "RESTful Web Services vs. 'Big' Web Services: Making the Right Architectural Decision," *Proc. 17th Int'l. Conf. World Wide Web*, ACM, 2008, pp. 805–814.
2. S. Vinoski, "RESTful Web Services Development Checklist," *IEEE Internet Computing*, vol. 12, no. 6, 2008, pp. 94–96.
3. L. Richardson and S. Ruby, *RESTful Web Services*, O'Reilly, 2007.
4. S. Allamaraju, *RESTful Web Services Cookbook*, O'Reilly, 2010.
5. R.T. Fielding, "REST APIs Must Be Hypertext-Driven," blog, 20 Oct. 2008; http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven.
6. L. Richardson, "Justice Will Take Us Millions of Intricate Moves," QCon presentation, 20 Nov. 2008; www.crummy.com/writing/speaking/2008-QCon/.
7. J. Algermissen, "Classification HTTP-based APIs," 2010, www.nordsc.com/ext/classification_of_http_based_apis.html.
8. I. Zuzak, I. Budiselic, and G. Delac, "Formal Modeling of RESTful Systems Using Finite-State Machines," *Web Engineering*, LNCS 6757, Springer, 2011, pp. 346–360.
9. S. Schreier, "Modeling RESTful Applications," *Proc. 2nd Int'l Workshop RESTful Design*, ACM, 2011, pp. 15–21.
10. R. Alarcón, E. Wilde, and J. Bellido, "Hypermedia-Driven RESTful Service Composition," *Service-Oriented Computing*, LNCS 6568, Springer, 2011, pp. 111–120.
11. J. Webber, S. Parastatidis, and I. Robinson, *REST in Practice*, O'Reilly, 2010.
12. S. Tilkov, "REST Litmus Test for Web Frameworks," blog, 4 Aug. 2010; www.innoq.com/blog/st/2010/07/rest_litmus_test_for_web_frame.html.

simple formal models to provide abstractions that encapsulate REST principles and steer the development process.

REST Principles

RESTful systems are based on message-oriented client-server interaction with the possibility of layered intermediaries and message caching by any component. On the Web, a browser application or machine-driven client interacts with servers through layers of caching proxies. Client-server interaction must be stateless, meaning that the client maintains and sends the session state with each request. On the Web, this is enabled by HTTP's statelessness. Furthermore, server components can extend client functionality by providing code-on-demand (COD) programs, such as JavaScript scripts.

The *uniform interface principle* further guides component behavior. Server functionality is exposed as uniquely identifiable resources. Client-server communication is based on the client receiving a resource's state information in the form of representations and sending those representations to the server to manipulate the resource's state. Requests and responses must be self-descriptive so that any component can process them. On the Web, resources are identified with URIs, whereas self-descriptiveness is achieved via standard representation media types and HTTP operations, headers, and status codes. Finally, clients should make requests only to resources identified with links in hypermedia representations of previously received responses. The "Related Research and Practice in REST Framework Development" sidebar presents more information about developing RESTful systems.

Framework-Driven Development Process

Figure 1 illustrates our view of a framework-driven development process for RESTful systems. The process explains developers' roles and

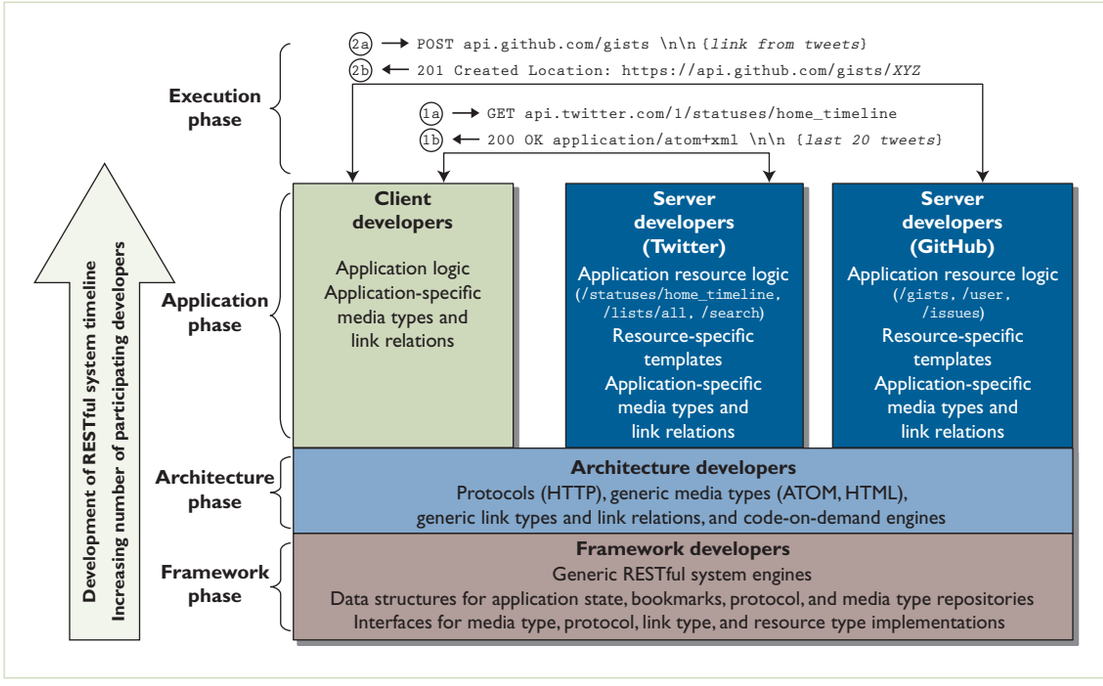

Figure 1. Framework-driven development process for RESTful systems. This process explains developers' roles and is structured into framework, architecture, and application phases. An example application based on the GitHub and Twitter APIs illustrates the application-level logic on both the client and server components.

is structured into *framework*, *architecture*, and *application* phases. The presented separation of concerns concentrates developer effort into individual domains of expertise and increases the framework elements' reusability. Specifically, application developers should focus on application goals and application-specific media types, architecture developers on technologies of a specific architecture, and framework developers on REST principles.

We illustrate this process with an example system for backing up Twitter tweets using GitHub *gists*, which are versionable snippets of text. A machine-driven agent periodically retrieves the user's tweets (steps 1a and 1b in Figure 1) using the Twitter API's `/statuses/home_timeline/` resource to fetch an Atom-formatted list of the last 20 tweets in the user's timeline. The agent then internally stores the retrieved tweets and periodically publishes them as a GitHub gist (steps 2a and 2b) using the GitHub API, which exposes a `/gists` collection resource for creating new gists.

The framework phase involves developing the core framework as architecture-independent modules required for implementing specific architectures and applications. These modules include both generic RESTful system engines that implement the processing flow in clients and servers, client application state and bookmark data structures, and the repositories for storing protocol and media type implementations. Furthermore, framework developers define the necessary component interfaces for the architecture and application layers.

The architecture phase involves implementing the technologies of specific RESTful architectures. Using the framework phase's interface definitions, architecture developers implement required application-independent protocols, media types, link types, and COD engines. In our example, this includes implementing the HTTP protocol, URI parsers, and the Atom media type that Twitter uses.

The application phase involves creating client and server application components using modules from the framework and architecture layers. Specifically, server developers implement resources bound to resource identifiers, such as those for accessing user favorites and statuses in the Twitter API. Similarly, client developers implement application-specific logic as either machine-driven agents or browsers. In both cases, we can view application-specific

logic as rules that process the obtained representations and select a hypermedia link for a user agent to follow. In the example, the user agent extracts, collects, and temporarily stores the tweets, and periodically follows a bookmarked link to the GitHub Gists resource. Furthermore, application developers implement application-specific media type processors that weren't defined during the architecture phase, such as the custom JavaScript Object Notation (JSON) format the GitHub API uses.

Guidelines and Framework Analysis

Here, we present guidelines for designing frameworks that incorporate REST principles and support the described development process. In parallel with describing guidelines, we analyze selected Web frameworks to determine their level of support for each guideline. We've chosen both server-side and client-side frameworks for different programming languages and paradigms that support building arbitrary RESTful systems: Webmachine (https://bitbucket.org/justin/webmachine), Jersey (http://jersey.java.net), Restfulie (http://restfulie.caelum.com.br), and RESTAgent (http://restagent.codeplex.com). First, we look at framework guidelines, which address system-wide guidance not specific to client- or server-based components.

Although we might consider some guidelines optional, frameworks that don't implement them will have either reduced functionality or reduced modifiability. Thus, application developers must re-implement modules from the architecture layer or even the framework layer. Table 1 summarizes the guidelines and framework analysis.

Framework Design Guidelines

First, frameworks should support system modifiability,[1] so developers can easily export, import, and change any architecture and application element definition — for example, they can define new protocol headers, introduce new resources, and change media type definitions. Well-defined module interfaces and a repository-oriented design for protocol and media type implementations help achieve such modifiability.

Frameworks should support the implementation of multiple application-level protocols, such as HTTP, and their simultaneous use. To support separation of concerns, a framework should promote the modularization of protocol implementation definitions, which should consist of the supported request operations (for example, HTTP `GET` and `POST`), response codes (such as HTTP `200` and `404`), and possible header names and values (for example, a BNF grammar defining the `Accept` header). Furthermore, a protocol implementation should contain protocol deserializers and serializers, which expose message elements from a byte stream, and vice versa. These message elements are control data, such as protocol operations and status codes; metadata, such as headers; and representation data. All analyzed frameworks are bound to HTTP or HTTP extensions, and the protocol implementation definitions are only partially modularized.

In addition, the framework should support the implementation of resource identifier namespaces, such as the URI namespace, as well as identifier templates. Thus, frameworks should implement template engines that parse and generate identifiers using templates and template variables. All the frameworks we analyze are bound to the URI namespace.

A framework should also support the implementation of different media types and their simultaneous use. For example, server components should be able to expose resources using both HTML and Atom media types, and user agents should be able to parse both types of representations. Media type implementation definitions should be modularized and consist of representation parsers for data and hypermedia links, serializers, and supported link types and relations. For instance, developers should be able to define both HTML and Atom media types as parsers that validate messages, build a DOM structure and extract the links, and define the link types and relations, such as `<a>` and `<img>` HTML link types, and the `<link>` link type and `self` link relation for Atom. All analyzed frameworks provide some support for extending the set of media types, but only RESTAgent supports the definition of link types.

Finally, frameworks should support content negotiation for choosing the media type for response representations, based on client preferences and server capabilities. For example, client developers should be able to express a preference for a custom JSON representation over an Atom representation, whereas servers supporting RSS and Atom media types should

Table 1. Analysis of existing frameworks for RESTful development.

Framework features	Webmachine	Jersey	Restfulie	RESTAgent	Comments
Languages	Erlang	Java	Ruby	C#	Restfulie is also available in Java and C#
Supported component types	Server	Server, client	Server, client	Client	
Framework guidelines					
Protocol implementation definition	○	○	○	○	Implementation definitions not modularized
Extensibility of supported protocols	–	–	–	–	Frameworks are bound to HTTP (and HTTP extensions)
Extensibility of supported namespaces	–	–	–	–	Frameworks are bound to URI namespace
Media type definition	○	○	○	+	Most frameworks have only parser and serializer, no link types
Extensibility of supported media types	+	+	+	+	
Content negotiation	+	+	+	–	
Client guidelines					
Generic client engine	n/a	–	○	○	Restfulie and RESTAgent: not completely automated
Flexible application state structure	n/a	–	–	○	RESTAgent: hashtable
Support for defining PLL, HLL, and ALL rules	n/a	–	–	–	
Bookmarks	n/a	–	–	+	
Link type definition support	n/a	–	–	○	RESTAgent: hard-coded set of state integrators – transfer, embed, replace, independent
Support for COD engines	n/a	–	–	–	
Server guidelines					
Generic server engine	+	+	+	n/a	
Resource types	○	○	○	n/a	No predefined ones
Mapping functions	–	–	–	n/a	
State machines for behavior	–	–	+	n/a	
Minting identifiers based on resource type	–	+	+	n/a	
Dispatching	+	+	+	n/a	Only supported for URI namespace

+: supported as described in the article; –: not supported; ○: partial support, with comments

automatically return an Atom representation. From the analyzed frameworks, only RESTAgent fails to support content negotiation.

Client-Oriented Guidelines

The client-oriented guidelines are based on a finite-state machine (FSM) formalization of RESTful client components.[5] In our opinion, an FSM-based approach is appropriate for modeling RESTful systems and offers well-known development abstractions.

Figure 2 presents the execution flow for client components and the development phase for each module. To satisfy statelessness, the client maintains the FSM's current state and also generates server requests using the *input*

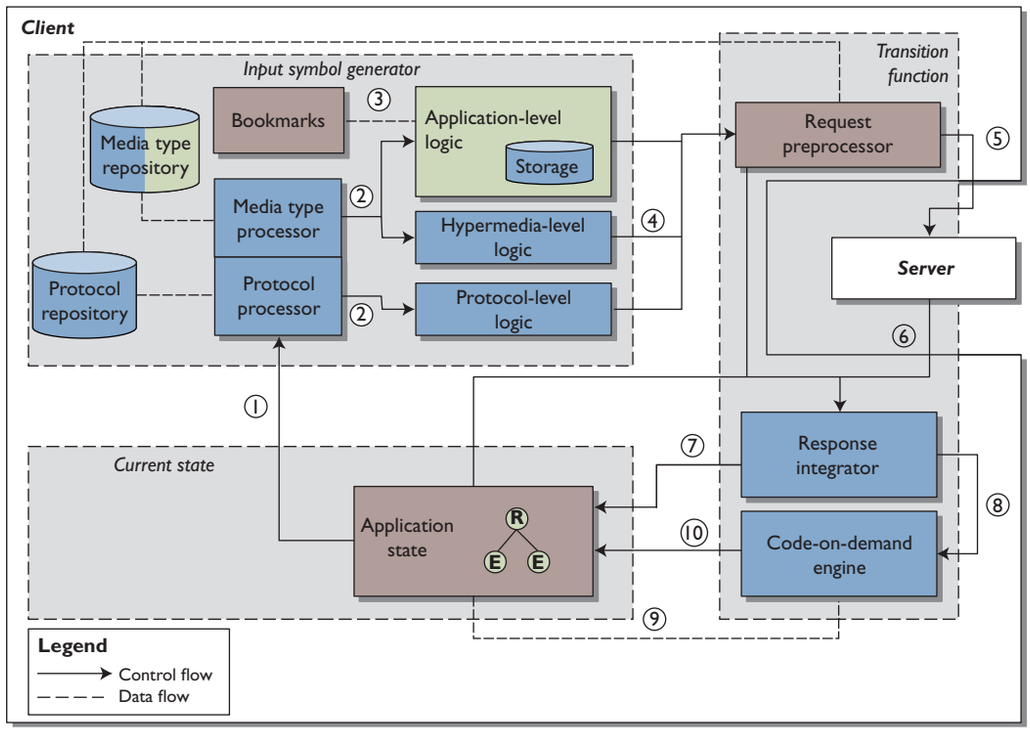

Figure 2. Generic RESTful client model and execution flow. Green elements represent modules implemented by client-side application developers; light blue elements those implemented by architecture developers; and brown elements those implemented by framework developers.

symbol generator. The *transition function* is divided between the client and server, where the server responds to requests while the client integrates responses into the next state.

The FSM's state is the set of resource representations present in the *application state*. The *protocol* and *media type processors* read these representations (step 1 in Figure 2). The protocol processor parses received messages into protocol message elements. The media type processor parses the representation data to extract the hyperlinks, where each link is defined with a resource identifier, link type, and link relation. The protocol and the media type processors pass the links to the *protocol-level logic* (PLL), *hypermedia-level logic* (HLL), and *application-level logic* (ALL) modules (step 2) to generate the next request. The PLL and HLL modules automatically bring the system into a steady state in which no outstanding requests exist, as with a completely loaded webpage. For example, PLL generates requests for handling resource redirection responses, such as `301` status codes in HTTP, whereas HLL generates requests for fetching embedded resource representations, such as HTML `<img>` links. When the system state is steady, ALL generates requests based on application-specific goals or user input. In a steady state, ALL might also bookmark links available in the application state (step 3) and reuse those links later as if they were still present. The *request preprocessor* validates and serializes the generated request (step 4) using the chosen link's link type. For example, the request processor validates that only HTTP `GET` requests are generated for HTML `<a>` links. Next, the request preprocessor sends the request to the server component (step 5). The *response integrator* uses the server's response representation (step 6) to construct the next application state (step 7). For example, for requests generated for navigational HTML `<a>` links, the response representation will replace the current application state. However, for requests generated for embedding `<img>` links, the response representation will be embedded in the current application state. Finally, if the received response is a COD script, a *COD engine* executes it (step 8)

based on the script's media type. The executing script can autonomously read (step 9) and change the application state (step 10).

Frameworks should provide an implementation of the described execution flow as a generic execution engine, whereas developers implement specific modules only. This inversion of control increases module decoupling and improves the modifiability of the framework and developed systems. Although Restfulie and RESTAgent provide a generic engine, they don't automate the complete processing flow.

The application state module should be flexible enough to accommodate various client types, such as browsers and machine agents. A directed graph would be an appropriate data structure, so nodes stand for representations and edges represent embedding links, with one representation marked as the root node. In a Web application, for example, an HTML document is the root node, and representations of embedded images and iframes are child nodes. Of the analyzed frameworks, only RESTAgent provides a hashtable-based application state data structure.

A framework should support the definition of PLL, HLL, and ALL as rules over the application state. Each rule checks for patterns in the application state; if a pattern is satisfied, the rule uses one of the links to determine the next request. Furthermore, the framework should execute rules according to the priority in RESTful systems: PLL first, HLL second, and ALL last. For example, for HTTP `301` redirect responses, first the PLL fetches the representation of the redirected resource, and then the HLL fetches embedded resources. The system is in a steady state when the PLL and HLL don't generate any new requests. Furthermore, the framework should enable the configuration of which PLL, HLL, and ALL rules are active, so that, for example, mobile agents can disable automatic fetching of embedded images. None of the analyzed frameworks use this approach for defining protocol, hypermedia, and application logic.

Frameworks should also allow the ALL to store any part of the application state and provide *bookmarks* storage for saving application entry-point links. In the Twitter-GitHub example, the agent uses internal storage to store received tweets before sending them to GitHub, while the link to the GitHub Gists resource is available as a bookmark. Of the analyzed frameworks, only RESTAgent provides bookmarking capability.

Frameworks should support the implementation of link types as tuples of a link type identifier — such as `<a>` in HTML — a request validation function, and a response integration function. The request preprocessor uses the request validation function to check whether PLL, HLL, and ALL may generate a given request from the current application state and for a specific link. The response integration function is called by the response integrator and produces the next application state by merging the response representation into the representation graph (step 7) based on the link type used to generate the request. These two functions together are the foundation of implementing REST's hypermedia principle. Only the RESTAgent framework supports different kinds of link types, but this set isn't extensible.

Finally, frameworks should support the definition of COD engines as functions that execute scripts with read-write access to the application state (step 9). None of the analyzed frameworks support COD engines.

Server-Oriented Guidelines

Because defining resources is one of the main tasks during server development, we base server-oriented guidelines on previous work on metamodels for RESTful server applications.[6]

Although resources are formally defined as "a conceptual, temporally varying mapping"[1] from resource identifiers to entities — such as user or tweet information — in most cases, an application consists of resources that have the same mapping and the same behavior. So, frameworks should support the definition of resource types, including their structural and behavioral parts. A named resource type describes the commonalities of a set of resources by defining their mapping function, their exposed data and hyperlinks, and how they react to particular protocol operations. A typed entity is a simple data object that has attributes and relationships with other entities. Supported by the framework, developers can use the type of entity to which a resource maps to define the data and hyperlinks the resource's representations offer. A resource won't usually offer all of an entity's internal details — for example, Twitter user resources won't expose

internal user-statistics information. Only server developers should use resource types; clients should be guided only by media types and link relations of received representations.[7]

Because we can distinguish resource types based on their content and relation to other resource types, frameworks should offer different structural resource types to minimize the effort of implementing concrete types in the application phase. An example of content-based distinction are list resources, such as the latest 10 tweets, or subresources. Whereas application developers can define resource types in all analyzed frameworks, none offer predefined ones.

Frameworks should also support different kinds of mapping functions. First, some resources map to the same entity as long as they exist — for instance, resources that map to specific tweets. Frameworks can support this constant mapping because mapping a resource to an entity and loading the entity from the persistence layer is possible without effort from application developers. Second, some resources have a varying mapping for every request, such as a random tweet resource. In such cases, application developers should be able to define the entity types these resource types map to and how concrete entities are calculated — by, for example, sorting tweets and randomly choosing one from the top 10. So, frameworks can automatically make the calculated entities available in the execution context upon receiving a request. None of the analyzed frameworks support defining mapping functions.

To support defining behavior, we suggest that each resource type defines a state machine that encodes the application-specific states that a resource of this type can be in. Each state defines the supported protocol operations and the behavior to be executed for valid requests. For example, defining resource behavior in the Restfulie framework is based on such state machines.

The framework should be able to create identifiers for resources based on their type's identifier template and a given entity. This enables the framework to support the rendering of hyperlinks in representations based on the relationship between two entities. Similarly, dispatching requires determining a given identifier's resource type. All the analyzed frameworks support template-based dispatching, while only Webmachine doesn't support minting identifiers from templates.

Figure 3 illustrates the execution flow that frameworks should implement as a generic engine for server-side processing. The *request parser* processes received requests (step 1 in Figure 3), parsing each request into message elements. Errors, such as invalid requests or application-level errors, will result in a response to the client (step 13) and in skipping other steps. The request parser forwards request message elements to the *dispatcher* (step 2), which locates the corresponding *resource type definition* (step 3) based on the request's resource identifier and sends it to the *request processor* (step 4). The request processor retrieves the definition's *mapping function* and *state machine* (step 5a) and, based on this mapping function, the mapped *entities* (step 5b). The *resource type state machine* asserts that the current state supports the requested protocol operation (step 6), while the *media type processor* deserializes the request body (step 7). The state machine defines the concrete behavior and invokes the required parts of the *business logic* (step 8), which in turn create and update the relevant entities (step 9) or connect to other systems for fulfilling the task. After processing the request, all information necessary for building the response is available (step 10). Finally, the *response builder* (step 11) builds the response representation using the *media type processor* (step 12) and sends it to the client (step 13). All analyzed frameworks implement server execution flows as a generic engine.

The development process and design guidelines we've presented provide practical advice for implementing or improving REST frameworks. Based on existing research on formal models, the described decomposition of client and server processing flows into generic modules enables better separation of concerns and greater system modifiability.

Although the Web frameworks we analyzed do support some of the presented guidelines, most guidelines aren't widely supported. In particular, frameworks should have better support for extending the supported protocols, defining hypermedia link types for managing application state, and using state machines to define resource behavior. Future work in this area

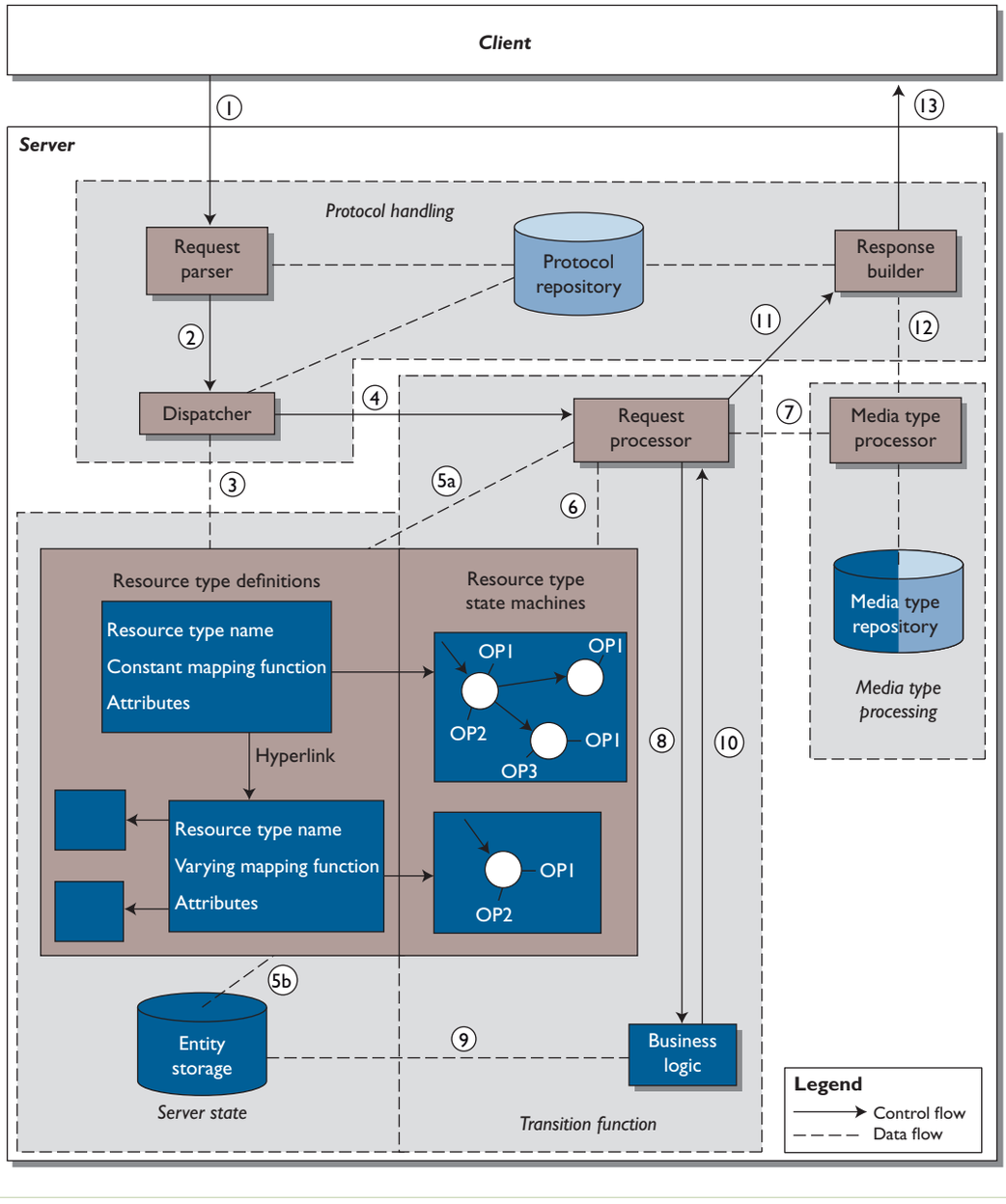

Figure 3. Generic RESTful server model and execution flow. Dark blue elements represent modules implemented by server-side application developers; light blue blue elements those implemented by architecture developers; and brown elements those implemented by framework developers.

includes defining guidelines for implementing intermediaries, enabling explicit caching support, and supporting machine-to-machine communication.

Acknowledgments

We acknowledge the support of the Ministry of Science, Education and Sports of the Republic of Croatia through the Computing Environments for Ubiquitous Distributed Systems (036-0362980-1921) research project. We're grateful for helpful feedback from Vedrana Jankovic from IN2; Ivan Budiselic, Dejan Skvorc, Miroslav Popovic, Klemo Vladimir, Marin Silic, Goran Delac, Zvonimir Pavlic, and Sinisa Srbljic from the Faculty of Electrical Engineering and Computing, University of Zagreb; and Christian Kollee and Bernd Krämer from the Faculty of Mathematics and Computer Science, University of Hagen.

References

1. R.T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, doctoral dissertation, Univ. of California, Irvine, 2000.
2. C. Pautasso, O. Zimmermann, and F. Leymann, "RESTful Web Services vs. 'Big' Web Services: Making the Right Architectural Decision," *Proc. 17th Int'l. Conf. World Wide Web*, ACM, 2008, pp. 805–814.
3. S. Vinoski, "RPC and REST: Dilemma, Disruption, and Displacement," *IEEE Internet Computing*, vol. 12, no. 5, 2008, pp. 92–95.
4. M. Broy, "Can Practitioners Neglect Theory and Theoreticians Neglect Practice?" *Computer*, vol. 44, no. 10, 2011, pp. 19–24.
5. I. Zuzak, I. Budiselic, and G. Delac, "Formal Modeling of RESTful Systems Using Finite-State Machines," *Web Engineering*, LNCS 6757, Springer, 2011, pp. 346–360.
6. S. Schreier, "Modeling RESTful Applications," *Proc. 2nd Int'l Workshop RESTful Design*, ACM, 2011, pp. 15–21.
7. R.T. Fielding, "REST APIs Must Be Hypertext-Driven," blog, 20 Oct. 2008; http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven.

Ivan Zuzak is a computer science PhD candidate and research assistant at the University of Zagreb, Faculty of Electrical Engineering and Computing (FER). His research interests include software architecture, REST, Web architecture, and inter-widget communication. Zuzak has an MEng in computer science from FER. Contact him at izuzak@gmail.com.

Silvia Schreier is a computer science PhD student and research assistant at the University of Hagen, Faculty of Mathematics and Computer Science. Her interests focus on model-driven development of resource-oriented applications. Schreier has a Diplom in computer science from the University of Erlangen-Nuremberg. Contact her at silvia.schreier@gmail.com.