Servent: A Unified SERVer-CliENT Semantics

MAX SNYDER, Harvard College, USA

This paper presents *Servent*, a domain-specific language for full-stack web development. Web development entails the maintenance of three software artifacts – a database, a server, and a website – each with a related, but perhaps transformed, representation of the same underlying object model. This redundancy is desired to ensure privacy of the data and efficiency of the server, but developers must maintain three distinct copies of the schema as well as transformations between. Servent is a JSX-like language to capture full-stack semantics.

Additional Key Words and Phrases: domain-specific language, full-stack development, database, server, website.

1 INTRODUCTION

Full-stack development is an approach to software engineering that separates an application into three software artifacts: a data model, a "back-end," and a "front-end." Consider a simple social media application where users can post messages to a global feed and "like" or "dislike" messages. Message metadata and each user's credentials are stored in a database. A back-end server acts as the intermediary between front-end website users and the database: messages are validated before insertion into the database to ensure correct credentials, and only the public data is fetched. This facilitates privacy of secure data as well as efficiency of serving data to many clients simultaneously.

However, developers of full-stack applications must maintain three or more representations of the same underlying object models. Consider possible representations of a message. The persistent object is stored in the database in a `Message` table, and user credentials are stored in a `User` table. The server must be able to receive objects with a message and credentials, and send objects with a message and no credentials. The client must be able to send objects with a message and credentials, and receive objects with a message and no credentials. In total, five representations are required to support posting and fetching. This number will continue to increase as new functionality is added.

Servent is a new JSX-like language (HTML + JavaScript) that provides a unified *full-stack semantics*: each object model is declared in one place, alongside all of its variations. These variations include an object's internal database representation, its back-end transformations, and its front-end formatting.

This paper offers the following contributions:

- **Part 1**: Introduction
  - a recapitulation of full-stack development: a database, a back-end, and a front-end.
  - a statement of the problem that Servent addresses.
  - a comparison of Servent with related work.
- **Part 2**: Implementation
  - a motivating example that introduces Servent's source code language.
  - a description of Servent's source code syntax.
  - a description of Servent's semantics and target code generation.
- **Part 3**: Conclusion
  - a summary and discussion of this paper's contributions.
  - a set of ideas for future work on the Servent implementation.

1.1 Full-Stack Web Development

Full-stack development involves a database, a back-end, and a front-end. In web development, the back-end is often called a "web server", and the front-end is often called a "website". Many frameworks exist for each artifact; this paper uses PostgreSQL, Spring Boot, and React JS, respectively.

1.1.1 Database.

A relational database system is a software artifact that provides an API, typically in SQL syntax, for inserting and selecting data. Data is partitioned into "tables," which correspond to objects like `Message`. Tables are partitioned into "columns," which correspond to object fields of a given type, including integral types like 32-bit integers and 64-bit longs and fixed- or variable-length strings. Each table is a set of "rows," which are each instances of an object with an entry in each column. Queries on a table include insertion, deletion, and selection and updates based on column value(s). The "join" query merges data between tables based on shared value(s) [Ramakrishnan and Gehrke 2003]. PostgreSQL is a relational database system [PostgreSQL Global Development Group 2022].

1.1.2 Back-End.

A back-end / "server," is a liaison between a relational database system and a front-end / "client". The server interacts with the database using the SQL API and interacts with the client using HTTP. Interfaces have been developed to enable SQL queries to be supported in other source languages. For instance, the Java Database Connectivity API allows SQL queries to be embedded in Java function annotations. The Jakarta Persistence API makes use of JDBC to correspond tables with Java classes using variable annotations. This enables queries to be expressed entirely in Java [Foundation 2022b]. Interfaces have been developed to enable HTTP objects to be serialized into host language objects. Java Spring Boot allows endpoints to be set up declaratively as Java functions [VMware-Inc. 2022].

1.1.3 Front-End.

A front-end / "client" for web is typically written in HTML/CSS/JavaScript, a trio of languages for content, design, and functionality, respectively. JavaScript provides mechanisms to send and receive HTTP requests and abstracts the delay using the "promise" construct [Madsen et al. 2017]. Each language in the trio was intended to be written in a designated file, but React JS enables HTML and JavaScript code to be embedded together in a unified "JSX" syntax [Platforms Inc. 2022].

1.2 Problem

Each software artifact in the full-stack architecture can be developed very easily using the declarative frameworks discussed above. However, the fragmentation of the object model leads to code dependencies between frameworks that are not captured semantically by one framework alone. Consider a modification to the database schema such as renaming a column. This requires cascading changes to the server and client code that may not be in sync. If the back-end is updated but the front-end is not, all code will still compile, but the HTTP requests will fail due to the name change.

1.3 Related Work

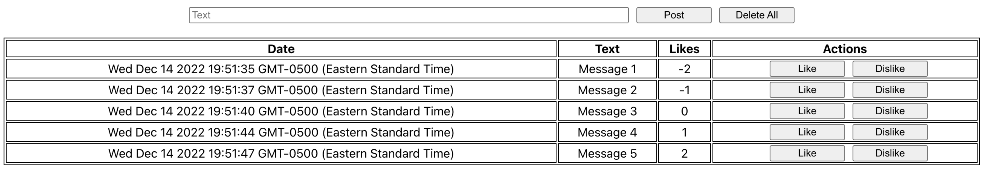
Servent is not the first framework to address this problem. Apache Thrift generates HTTP boilerplate code for many back-ends and front-ends. However, additional code must be written to connect the generated back-end code to the database as well as to transform the generated front-end code to be displayed on the client, which still results in redundant representations [Foundation 2022a]. WebDSL goes further by incorporating elements like validation into the object model, but enforces the use of a new language rather than of embedding into the back-end or front-end [Visser 2008].

2 IMPLEMENTATION

The implementation of Servent is located on GitHub [Snyder 2022].

2.1 Example

The following is the Servent implementation of the social media application from the introduction:


Text Post Delete All

Date	Text	Likes	Actions
Wed Dec 14 2022 19:51:35 GMT-0500 (Eastern Standard Time)	Message 1	-2	Like Dislike
Wed Dec 14 2022 19:51:37 GMT-0500 (Eastern Standard Time)	Message 2	-1	Like Dislike
Wed Dec 14 2022 19:51:40 GMT-0500 (Eastern Standard Time)	Message 3	0	Like Dislike
Wed Dec 14 2022 19:51:44 GMT-0500 (Eastern Standard Time)	Message 4	1	Like Dislike
Wed Dec 14 2022 19:51:47 GMT-0500 (Eastern Standard Time)	Message 5	2	Like Dislike

```
<MAIN>
    <TABLE name="Table1">
        <Column1 public long default={() => Date.now()} label="Date">
            {(l) => new Date(l).toString()}
        </Column1>
        <Column2 public string condition={(s) => s.length !== 0} label="Text" />
        <Column3 public int default={() => 0} label="Likes" />
        <Column4 void label="Actions">
            <>
                <button onClick={Table1.PATCH.Column3.INC}>
                    Like
                </button>
                <button onClick={Table1.PATCH.Column3.DEC}>
                    Dislike
                </button>
            </>
        </Column4>
    </TABLE>
    <br />
    <input onChange={Table1.POST.Column2.onChange}
           placeholder="Text"
           value={Table1.POST.Column2.value} />
    <button onClick={Table1.POST}>
        Post
    </button>
    <button onClick={Table1.DELETE}>
        Delete All
    </button>
    <br />
    <br />
    {Table1.GET.LENGTH > 0 && Table1.GET}
</MAIN>
```

The source code is in a JSX/HTML/XML-like syntax. Code is written as tags nested under the MAIN tag, and can take the form of a TABLE definition or an arbitrary JSX expression. Tables are collections of column definitions specifying properties about each column. The most obvious property is the (optional) column formatting, which is a JSX function or expression nested under each column definition. Other properties are specified as HTML "attributes," or key-value pairs with optional value. One deviation from JSX is the addition of Servent *Hooks*, not to be confused with React JS Hooks, which are references to table definitions that facilitate the linking of HTML elements – buttons, inputs, and tables – to HTTP functionality – patches (e.g. likes), posts, deletes, and gets.

2.2 Syntax

$$\begin{array}{rcll}
\mathit{Program} & ::= & \texttt{<MAIN>}\ [\mathit{Expression}]^*\ \texttt{</MAIN>} & \text{Program} \\
\mathit{Expression} & ::= & \mathit{Code} \mid \mathit{String} \mid \mathit{Table} \mid \mathit{Tag} \mid \mathit{Text} & \text{Expression}
\end{array}$$

A *Program* is a collection of *Expression*s nested under the *MAIN* tag.
An *Expression* is either JSX *Code*, a quoted *String*, a *Table* definition, a JSX *Tag*, or plain *Text*.

$$\begin{array}{rcll}
\mathit{String} & ::= & \text{"}\ \mathit{Text}\ \text{"} & \text{String} \\
\mathit{Text} & ::= & [.]^* & \text{Text}
\end{array}$$

A *String* is a quoted string of characters used as HTML attributes.
A *Text* is an arbitrary string of characters.

$$\begin{array}{rcll}
\mathit{Code} & ::= & \{\ [\mathit{Hook} \mid \mathit{JavaScript}]^*\ \} & \text{Code} \\
\mathit{Hook} & ::= & \mathit{Text}.\texttt{DELETE} & \text{Delete Hook} \\
 & \mid & \mathit{Text}.\texttt{GET} & \text{Get Hook} \\
 & \mid & \mathit{Text}.\texttt{GET.LENGTH} & \text{Length Hook} \\
 & \mid & \mathit{Text}.\texttt{PATCH}.\mathit{Text}.\texttt{DEC} & \text{Patch Hook (decrement)} \\
 & \mid & \mathit{Text}.\texttt{PATCH}.\mathit{Text}.\texttt{INC} & \text{Patch Hook (increment)} \\
 & \mid & \mathit{Text}.\texttt{POST} & \text{Post Hook} \\
 & \mid & \mathit{Text}.\texttt{POST}.\mathit{Text}.\texttt{onChange} & \text{Post Hook (Setter)} \\
 & \mid & \mathit{Text}.\texttt{POST}.\mathit{Text}.\texttt{value} & \text{Post Hook (Getter)} \\
\mathit{JavaScript} & ::= & \mathit{Exp}_{JS} & \text{JavaScript Code}
\end{array}$$

A *Code* is a sequence of *Hook*s and *JavaScript*s.

- A *Hook* is one of the following references to a table definition:
  - A *Delete Hook* must be on an HTML button's `onClick` attribute, and deletes a table's data.
  - A *Get Hook* is an HTML representation of a table.
    - A *Length Hook* is an integer representation of the number of rows from the *Get Hook*.
  - A *Patch Hook* must be on an HTML button's `onClick` attribute, and updates a cell's data.
    - A *Decrement Hook* decreases a cell's value by 1.
    - A *Increment Hook* increases a cell's value by 1.
  - A *Post Hook* must be on an HTML button's `onClick` attribute, and posts the value from the Setter and Getter. The Setter and the Getter must be on an HTML button's `onChange` and `value` attributes, respectively, and provides one field of a new row posted by the *Post Hook*.
- A *JavaScript* is an arbitrary string of JavaScript code, described by λ_{JS} [Madsen et al. 2017].

$$\begin{array}{rcll}
\mathit{Tag} & ::= & <\mathit{Text}\ [\mathit{Attribute}]^*\ \texttt{/>} & \text{Tag (without children)} \\
 & \mid & <\mathit{Text}\ [\mathit{Attribute}]^*\ >\ [\mathit{Expression}]^*\ </\ \mathit{Text}\ > & \text{Tag (with children)} \\
\mathit{Attribute} & ::= & \mathit{Text} & \text{Attribute (without value)} \\
 & \mid & \mathit{Text} = \mathit{Code} \mid \mathit{Text} = \mathit{String} & \text{Attribute (with value)}
\end{array}$$

A *Tag* an HTML element with possibly non-empty collections of attributes and nested children.
An *Attribute* is a key-value pair with an optional *Code* or *String* value.

$$\begin{array}{rll} \textit{Table} & ::= \texttt{<TABLE name=}\,\textit{String}\,\texttt{>}\ [\textit{Column}]^{*}\ \texttt{</TABLE>} & \text{Table} \\ \textit{Column} & ::= \textit{Tag} & \text{Column} \end{array}$$

A *Table* is a named collection of columns.
A *Column* is a named HTML tag with the following key-value attributes:

- **Access** Attribute (optional, no value):
  - `public`: the column is present in PATCH/POST requests, but not GET requests.
  - `private`: the column is present in PATCH/POST and GET requests.

  If the access attribute is not present, the default value is `public`.
- `condition` Attribute (optional): a JSX pre-condition function for a successful POST request.
- `default` Attribute (optional): a JSX function used if a value is missing from a POST request.
- `label` Attribute (optional): a *String* representing the column in the HTML table header. If the label attribute is not present, the default value is the column name (e.g. `Column1`, `Column2`).
- **Type** Attribute (required, no value):
  - `int`: a 32-bit signed integer.
  - `long`: a 64-bit signed integer.
  - `string`: a variable-length string of characters.
  - `void`: not used by the database or server, and thus used only for display in GET requests.

A *Column*'s children represents the formatting for the column. If the child is a JSX function, the column value will be passed dynamically as the first argument. Otherwise, the JSX code is used as-is to format the column, except *Patch Hooks* are replaced by their corresponding patch requests.

2.3 Semantics

Servent is a transpiler, or a source-to-source compiler, thus the semantics of the system can be described with respect to the target JSX code for the front-end and JavaScript code for the back-end.

2.3.1 Front-End Semantics.

Much of the front-end JSX code is directly transported as-is from the children of the `MAIN` tag, excluding table definitions. Additional code is inserted to set up the HTTP endpoints for each hook and to replace the hooks in-place with their corresponding action or value. Moreover, some column properties manifest themselves as additional code on the front-end. The condition property is used to validate a column value before a POST request, aborting the request if the function returns false. The default property is used in the POST request body if no Setter and Getter hooks are present for a column. The label and formatting properties are used while displaying the table by the Get hook.

2.3.2 Back-End Semantics.

The back-end Java code is auto-generated by the transpiler. Each table manifests itself as a Spring Boot microservice with several corresponding target code files. Each microservice has a "controller" to set up HTTP endpoints, a "row" to link the object model to the database using JPA, a "repository" to provide database operations using JPA, and a "service" to do work requested by the controller. Moreover, each supported request, excluding DELETE, has a corresponding data transfer object, or "DTO," that enables the client to send request-specific information. The GET DTO has `public` fields, the POST DTO has `public` and `private` fields, and the PATCH DTO describes field updates.

```
@RestController
@RequestMapping("/Table1")
public class Table1Controller {
    @Autowired
    Table1Service service;

    @DeleteMapping(value = "/DELETE")
    public ResponseEntity<Response<Void>> delete() {
        service.delete();
        return Response.ok();
    }

    @GetMapping(value = "/GET")
    public ResponseEntity<Response<List<Table1GetDTO>>> get() {
        List<Table1GetDTO> dtos = service.get();
        return Response.ok(dtos);
    }

    @PatchMapping(value = "/PATCH")
    public ResponseEntity<Response<Void>> patch(@RequestBody Table1PatchDTO dto) {
        service.patch(dto);
        return Response.ok();
    }

    @PostMapping(value = "/POST")
    public ResponseEntity<Response<Void>> post(@RequestBody Table1PostDTO dto) {
        service.post(dto);
        return Response.ok();
    }
}
```

The controller creates HTTP endpoints using Spring Boot annotations like `RequestMapping` and `DeleteMapping` to yield a URL like `localhost:8080/Table1/DELETE`. The controller is not aware of the database model, referring to objects using DTOs, as the Service performs database operations.

```
@Entity
@Table(name = "Table1")
public class Table1Row implements Comparable<Table1Row> {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    public UUID id;

    @Column(name = "idPublic")
    public UUID idPublic;

    @Column(name = "Column1")
    public Long column1;

    @Column(name = "Column2")
    public String column2;

    @Column(name = "Column3")
    public Integer column3;

    @CreationTimestamp
    private Timestamp creationTimestamp;

    @Override
    public int compareTo(Table1Row row) {
        return this.creationTimestamp.compareTo(row.creationTimestamp);
    }
}
```

The row, or "entity," declares the database schema for the table using an annotated Java class. Each row additionally has two identifiers, an internal `id` and an exposed `idPublic` for use in GET and PATCH requests. Moreover, each row has a `createionTimestamp` for sorting data in GET requests.

```
public interface Table1Repository extends JpaRepository<Table1Row, UUID> {
    Table1Row findByIdPublic(UUID idPublic);
}
```

The repository provides operations like `deleteAll()`, `findAll()`, and `save()`, corresponding to DELETE, SELECT, and INSERT/UPDATE SQL queries, and custom queries like `findByIdPublic`.

```
@Service
public class Table1Service {
    @Autowired
    Table1Repository repository;

    public void delete() {
        repository.deleteAll();
    }

    public List<Table1GetDTO> get() {
        return Table1GetDTO.list(repository.findAll());
    }

    public void patch(Table1PatchDTO dto) {
        repository.save(dto.patch(repository.findByIdPublic(dto.id)));
    }

    public void post(Table1PostDTO dto) {
        repository.save(dto.row());
    }
}
```

The service injests DTOs from the controller, performs transformations from DTOs to rows, performs database operations using the repository, and performs transformations from rows to DTOs.

3 CONCLUSION

3.1 Discussion

Servent upends the development process for full-stack applications. The developer must now first abstractly consider what the "schema" is for the application in order to create the table and column definitions. This schema can evolve over time, starting with most attributes empty. Then, the developer must consider which subset of table and column definitions with which a user will directly interact, and embed the corresponding hooks into the front-end code. This separation of concerns between data and functionality is typically implicitly enacted by the developer, but Servent reifies this paradigm in its semantics in order to eliminate redundant boilerplate code. This empowers the JPA and Spring frameworks to fully realize their declarative nature across the stack.

3.2 Future Work

- **Custom Patch Hooks**:
  Currently, the only Patch hooks are increment and decrement. The user should be able to specify custom Patch hooks as transformations from integers to integers or strings to strings. This would likely take the form of custom JavaScript functions located in a designated file.
- **Custom Hooks**:
  Currently, all server endpoints correspond to CRUD (create, read, update, delete) database operations. The user should be able to create custom endpoints and the corresponding hooks. This would likely take the form of custom Java/Spring functions located in a designated file.
- **Standard Library for HTML Elements**:
  Currently, hooks can only be embedded in `button` and `input` elements. Since many other HTML elements have similar `onClick`, `onChange`, and `value` attributes (e.g. checkboxes), so it would be desirable to have standard library of compatible Servent embeddings into HTML.
- **Table Joins**:
  Currently, table definitions may only be self-referential. Consider an extension to the social media application where the user must authenticate by providing their credentials in each POST or PATCH request. Credentials are located in a separate `User` table, so it would be desirable to functionally link two tables by using a custom hook or creating a new Join hook.

ACKNOWLEDGMENTS

Thank you to Liam McInroy for brainstorming sessions, Lynx FinHealth for introducing me to PostgreSQL, React JS, and Spring Boot, and Professor Amin for another great semester in CS252R.

REFERENCES

Apache Software Foundation. 2022a. *Apache Thrift.* Apache Software Foundation. https://thrift.apache.org/

Eclipse Foundation. 2022b. *Jakarta Persistence.* Eclipse Foundation. https://projects.eclipse.org/projects/ee4j.jpa

Magnus Madsen, Ondřej Lhoták, and Frank Tip. 2017. A Model for Reasoning about JavaScript Promises. In *Proc. ACM Program. Lang.* Association for Computing Machinery, New York, NY, USA, 86:1–86:24. https://dl.acm.org/doi/pdf/10.1145/3133910

Meta Platforms Inc. 2022. *React.* Meta Platforms Inc. https://reactjs.org/

The PostgreSQL Global Development Group. 2022. *PostgreSQL.* The PostgreSQL Global Development Group. https://www.postgresql.org/

Raghu Ramakrishnan and Johannes Gehrke. 2003. *Database Management Systems, Third Edition.* McGraw-Hill, 1221 Avenue of the Americas, New York, NY 10020. https://pages.cs.wisc.edu/~dbbook/

Max Snyder. 2022. *Servent.* GitHub. https://github.com/maxsnyder2000/servent

Eelco Visser. 2008. WebDSL: A Case Study in Domain-Specific Language Engineering. In *GTTSE 2007, LNCS 5235.* Springer-Verlag, Berlin Heidelberg, 291–373. https://eelcovisser.org/publications/2007/Visser07.pdf

VMware-Inc. 2022. *Spring.* VMware Inc. https://spring.io/