A Self-Hosting Point-and-Click Editor for Any Functionally-Specified GUI-DB Application

Michael A. White

April 2026

Abstract

I describe the architecture of a declarative full-stack web framework and self-hosting point-and-click editor for any functionally-specified graphical database-backed (GUI-DB) application that reduces application development to its true essential complexity in a single object-oriented language with only small parts in a trivial transpiled embedded functional language using a multi-layered Single Page Application architecture with a functional-reactive view framework, a relational database, an industrial-strength object-relational mapper, and several "magic" components. There is no sacrifice in performance compared to a hand-coded application and all industry standard web application functionality can be provided using a comprehensive plugin system.

Contents

1 Acknowledgments

Thanks to my parents for believing in me and my father for letting me use a word processor and later his computer in the '90s and reading the earliest short proofs by email. Thanks also to my aunt and uncles, cousins, first cousins once removed (up), and second cousins for reading early versions in the '90s and tolerating indefinite requests for related action.

Thanks to philosopher Daniel Dennett for tolerating an email with the short proof that was outside his specialty and possibly mentioning the idea to others. Thanks to Tufts University computer science professor Norman Ramsey for continuing to correspond on the idea and instilling of proper interface discipline in COMP40 and an appreciation for functional programming.

Thanks to the Dimagi.com dev team for many educational software debates, and a great open source form and case solution that may be adapted for future versions of this paper: Jon Jackson, Cory Zue, Biyeun Buczyk, and Daniel Roberts.

Thanks to David Pincus, my surname folk etymology friend, for renting to me while I wrote the final version of this paper and critiquing my "in object-oriented languages" claim when I initially came up with the idea.

Thanks to the Massachusetts Institute of Technology for letting me walk the Infinite Corridor twice.

2 Introduction

Beginning in 1995, after writing some server-side web applications in C using a web application server, object-relational mapper (ORM), and template engine, and after being exposed to research on functional-reactive view libraries, and additionally building on the key insight that a truly seamless graph-based access control (GBAC) library for a Python[1] object-relational mapper that delegates all privilege calculation to the database but allows privilege definition in Python would allow fully declarative multi-layer client-server architectures to be securely extended to non-trivial applications for the first time, I have undertaken a concerted attempt to identify ultimate best practices in scalable interactive GUI-DB application development (including GUI or backend-only applications) through the lens of interactive web applications, knowing that they can also be made into desktop and mobile applications, and abstract them into an easy-to-use web framework and, if possible, a point-and-click

[1] `http://python.org`

editor that outputs code that uses the framework, in a way that fully follows SOLID [2] design principles.

Contrary to widespread belief, the creation of complex interactive web applications is revealed to be possible with a point-and-click editor that is not merely an IDE that recapitulates the low-level abstract syntax tree of any application, without any sacrifice in performance over a hand-coded version of the application. Many aspects of the web development industry, which I claim reached a state of relative perfection (minus the framework and editor) in 2016 to astute observers,[3] have been commented on or predicted. The editor ends up being self-hosting by employing a high degree of what is sometimes referred to as "magic" in the software engineering profession, with all model and view definitions able to be stored in a structured manner in the database, though it is recommended to continue storing the canonical application definition as text, making the editor's self-hosting nature a sort of pass-through or verification-only self-hosting that is considered a forcing function to achieve a fully declarative editor with end-user branching version control rather than an inherently desirable property. The framework can be written in either JavaScript[4] with a minimal embedded transpiled functional language for some parts or any full functional language that compiles to JavaScript. The Python backend is mostly hidden by JavaScript through a bridge in the final product.

The design satisfies the following desiderata:

- Implemented in one or two object-oriented languages
- No or minimal build time (interpreted languages)
- Instant deployments
- Web, mobile, or desktop applications
- Web or native widgets for mobile and desktop applications
- Fully follows the UNIX philosophy at all levels
- Able to be monetized

[2] `http://en.wikipedia.org/wiki/SOLID`

[3] with the release of React Router version 3

[4] `http://javascript.org`

3 A Declarative Multi-Layer Full-Stack Single Page Web Application Architecture

The first step in the creation of a higher-level web framework is the identification of a standard web application architecture for interactive Single Page Application applications (applications that do not reload the page in order to display a new view) with declarative APIs in object-oriented languages, JavaScript in the browser (the frontend) and Python on the server (the backend).

The architecture is based on the use of a relational database[2] queried using SQL as representing the essential complexity of data storage, with NoSQL databases considered unnecessary for almost all use cases and relational databases able to be scaled just as well using horizontal scaling techniques.

- Backend (Python):
  1. Web framework (Flask[5])
  2. An industrial-strength unit-of-work object-relational mapper (ORM) that fully exposes the SQL language in a Pythonic API where the expression abstract syntax tree (AST) is available to inspect and modify, and additionally featuring events, hybrid properties[6] (model properties defined using the ORM expression language that can be referenced in a query), single and multi-table inheritance through class inheritance, session and query compile extension hooks, a file attachment extension, and an aggregate column extension (SQLAlchemy[1][7] with SQLAlchemy-Utils[8])
  3. A JSON API REST API generator (Flask-Restless [9])
  4. Custom API endpoints (allowing only ORM and external HTTP requests library)

[5] `http://flask.palletsprojects.com`
[6] `https://docs.sqlalchemy.org/en/20/orm/extensions/hybrid.html`
[7] `http://sqlalchemy.org`
[8] `https://sqlalchemy-utils.readthedocs.io/en/latest/`
[9] `https://github.com/mrevutskyi/flask-restless-ng`

[2] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[1] Michael Bayer. Sqlalchemy. In Amy Brown and Greg Wilson, editors, *The Architecture of Open Source Applications Volume II: Structure, Scale, and a Few More Fearless Hacks*. aosabook.org, 2012.

- Frontend (JavaScript):
    5. Relational REST[3] Models with model and collection events (Backbone[10] models with Backbone-Relational[11])
    6. A functional-reactive view framework that uses JavaScript classes as components with JSX (XML-like markup that allows arbitrary embedded expressions in JS) for the component tree, with a virtual DOM and component lifecycle methods for short-circuiting rendering based on data (React[12]). *Implementation note:* "Function components"[13] that work using a global render state hack most likely considered an anti-pattern or unnecessary compared to standard class-based components with higher-order components.
    7. Dynamic isomorphicURL router (React Router (v3.x)[14]). An isomorphic frontend router is one that enables a one-to-one mapping between child routes and child components for applications like an email application with a sidebar and inbox and compose routes.
    8. CSS modules[15] - automatic global scoping of style imports in JavaScript (generation of anonymous class names and ids scoped to the file name available as a variable in JavaScript, for conflict-free use of multiple stylesheets in an application)

In addition, many applications make use of an optional widget library or toolkit that provides a few dozen standard UI widgets, which could be considered the ninth layer: React Bootstrap[16] or Material UI[17] for web applications, React Native for mobile applications, and React GTK or React Native Desktop QT for desktop applications

The general architecture that follows has become widespread, with several available implementation choices for each layer in multiple programming languages, though some choose a less powerful object-relational mapper that is easier for beginners to use, frontend REST models are sometimes omitted and CSS modules is not universal.

[10] `http://backbonejs.org`
[11] `http;//backbonerelational.org`
[12] `http://reactjs.org`
[13] `https://react.dev/reference/react/Component`
[14] `https://github.com/remix-run/react-router`
[15] `https://github.com/gajus/react-css-modules`
[16] `https://react-bootstrap.netlify.app/`
[17] `https://mui.com/material-ui/`

[3] Roy Thomas Fielding. *Architectural Styles and the Design of Network-Based Software Architectures*. PhD thesis, University of California, Irvine, Irvine, CA, USA, 2000. PhD Dissertation.

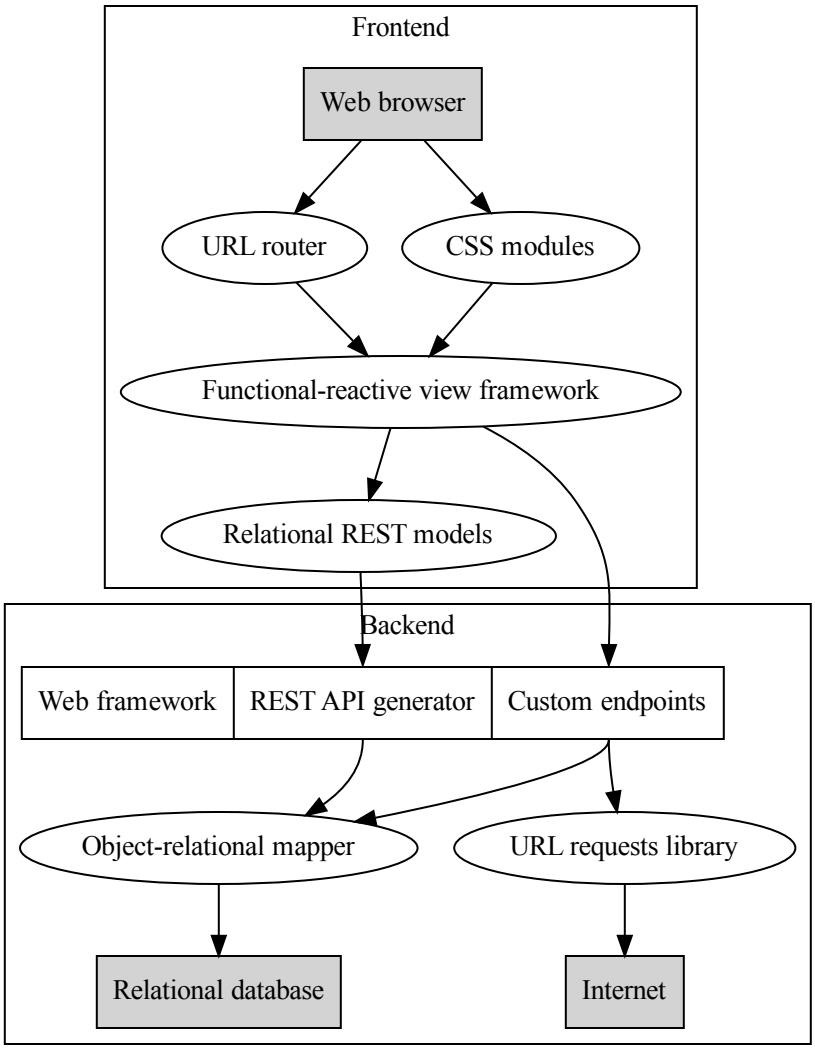


Figure 1: Declarative web application architecture

An example application looks like this (note that some code is omitted):

app.py

```
from flask import Flask
from flask_restless import APIManager
from flask_sqlalchemy import Database
from sqlalchemy import Column, Integer, Text
from sqlalchemy.orm import relationship

app = Flask(__name__)
db = Database(app)

class User(db.Model):
    __tablename__ = 'user'

    id = Column(Integer, primary_key=True)
    username = Column(Text, nullable=False, unique=True)

class Post(db.Model):
    __tablename__ = 'post'

    id = Column(Integer, primary_key=True)
    title = Column(Text, nullable=False)
    content = Column(Text, nullable=False)
    author_id = Column(Integer, ForeignKey('user.id'), nullable=False)
    author = relationship(User, backref='posts')

api_manager = APIManager(app, session=session)
api_manager.create_api(User, collection_name='users')
api_manager.create_api(Post, collection_name='post')

app.run()
```

main.js

```
import React from 'react';
import {Router, Route, Link} from 'react-router';
import Backbone from 'backbone';
import 'backbone-relational';
```

```
const User = Backbone.RelationalModel.extend({
    type: 'user',
    urlRoot: '/users'
});

const UserCollection = Backbone.Collection.extend({
    model: User
});

const Post = Backbone.RelationalModel.extend({
    type: 'post',
    relations: [{
        type: Backbone.HasOne,
        key: 'author',
        relatedModel: User
    }]
});

const App = React.createClass({
    componentDidMount() {
        const authors = new UserCollection();
        this.setState({
            authors: authors
        })
        authors.on('sync', this.update.bind(this))
        authors.fetch()
    },
    render() {
        const authors = {this.state};
        return <ul>
            {authors.map((author) => {
                return <li>
                    Author: {author.get('username')}
                    {author.get('posts').map((post) => {
                        return <Link to="post" params={{
                            id: post.get('id')
                        }}>
                            {post.get('title')}
```

```
                        </Link>;
                    })}
                </li>;
            })}
        </ul>;
    }

})

const Post = React.createClass({
    render() {
        // post view
    }
});

const Router = <Router>
    <Route name="home" path="/" component={App} />
    <Route name="post" path="/post/:id" component={Post} />
</Router>;

React.render(
    <Router/>
    document.getElementByID('container')
);
```

4 A High-Level Single Page Application Web Framework

Several "magic" components enable even more concise description of applications in a web framework that eliminates all forms of "boilerplate". It can be shown that the six-layer architecture (see 3) plus the first two magic components, seamless GBAC for ORMs and declarative data-loading for the functional-reactive view framework, represent the essential complexity of interactive database-backed applications, in combination with a plugin system (see 4.6).

The PostgreSQL[18] relational database is recommended for its no-compromises engineering, but any relational database with a JSON column type (see 4.3 below) and concurrent index creation (see 4.8.2) can in theory be used, which also includes MySQL, SQL Server, and Oracle. (However, one feature, JSON tables (see 4.3), should ideally be implemented as a DDL query rewriting extension to follow the UNIX philosophy, which would require the use of PostgreSQL). The latest relational databases are assumed to have eliminated the need for any NoSQL or NewSQL databases as long as sufficient horizontal scaling capability exists.

4.1 Seamless graph-based access control for object-relational mappers

Seamless graph-based access control, in which the CRUD privileges for each model are defined declaratively in terms of the model class and joins of related entities in terms of a user model for the current user and then automatically applied by an ORM extension for all models referenced in a query, enables non-trivial applications to use only a REST API generator for default data requirements. In addition, abstracting out access control in a flexible way makes the application code much more concise and makes it easier to prevent security lapses. Any more specific access control mechanism, such as role-based access control, can be achieved by modeling the access control data naturally.

SQLAlchemy GBAC[19] : Seamless row and cell-level graph-based access control for SQLAlchemy with no additional queries or duplicate joins that delegates all privilege calculation to the database in a transaction-compliant manner for all CRUD operations, including insert. *Implementation:* Surface column elements in expressions and add a set of declaratively-defined privilege calculations for each column to the query, supporting single and multi-table inheritance[20] through class inheritance, with a default filter mode that only returns visible rows and a validate mode that errors if any queried rows are not visible, and a query option for skipping privilege checking for specific columns or tables.

Note: PostgreSQL supports row-level graph-based access control without any modifications to the ORM through its Row Level Security[21] feature but does not currently support cell-level graph-based access control. There are some advantages to implementing access control at the ORM level, namely:

[18] `http://postgresql.org`
[19] `http://gitlab.com/mawhite/sqlalchemy_gbac` (private repo)
[20] `https://docs.sqlalchemy.org/en/20/orm/inheritance.html`
[21] `https://www.postgresql.org/docs/current/ddl-rowsecurity.html`

- Easy inheritance through classes
- The ability to query the database manually without access control limitations if one has the database credentials (this could be either an advantage or a disadvantage depending on the use case)
- Works with any database
- Automatic maintenance of correct privileges in the ORM's instance cache to do other operations with including when using a nested transaction that modifies the data used to calculate the privilege rather than manually updating a cached row in the application.

An example looks like this:

```
from sqlalchemy import Column, Boolean, Integer, ForeignKey
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy_gbac import GBACMixin, expression_property

Base = declarative_base(cls=GBACMixin)

class Todo(Base):
    id = Column(Integer, primary_key=True)
    is_complete = Column(Boolean, default=False)
    is_locked = Column(Boolean, default=False)
    owner_id = Column(Integer, ForeignKey('user.id'))
    owner = relationship(User)

    @classmethod
    def join_entities(cls, target):
        user_alias = aliased(User)

        return (
            ('user_alias',
                ('outerjoin', user_alias,
                 user_alias.id==target.owner_id)),
        )

    @expression_property
    def is_insertable(cls, target, session):
```

```
        user_id = session.info['current_user'].id

        return target.owner.id == user_id

    @expression_property
    def is_updateable(cls, target, session):
        user_id = session.info['current_user'].id

        return (target.owner.id == user_id) & ~target.is_locked

    @expression_property
    def is_selectable_(cls, target, session, user_alias):
        user_id = session.info['current_user'].id

        # A contrived alias example
        return user_alias.id == user_id

    @expression_property
    def is_deleteable(cls, target, session):
        return target.is_complete

class Assignee(Base):
    __tablename__ = 'assignee'
    __default_privileges__ = ('insert',)

    id = Column(Integer, primary_key=True)
    todo_id = Column(Integer, ForeignKey('todo.id'))
    value = Column(Boolean, default=True, info={
        # example cell-level privilege
        'select': lambda cls: cls.is_value_selectable
    })
    _is_value_selectable = Column(Boolean, default=False)

    todo = relationship(Todo, backref='assignees')

    @expression_property
    def is_selectable_(cls, target):
        return target.value
```

```
@expression_property
def is_value_selectable(cls, target):
    return target._is_value_selectable
```

4.2 Automatic asynchronous data loading for functional-reactive views

In traditional functional-reactive applications, data is manually loaded from an API in a lifecycle method (such as React's `componentDidMount()`) with event handlers for data changes being attached manually. Instead, data can be automatically loaded from declaratively-specified data requirements for each component in terms of the passed component props or active route parameters. In addition, data requirements can be automatically parsed from the view code.

React JSON API[22]: Declarative asynchronous data loading for the React functional reactive view library that parses the data requirements from relational REST model property references in the view code with optional frontend URL router integration and a comprehensive caching system *Implementation:* Iteratively string search the view definition source for attribute and relation references while up-propagating conditionals referencing the URL information or component params to generate a declarative description of the data requirements suitable to be passed to an explicit declarative data loading library that takes a function of the matched route information and the component props. Data is loaded into Backbone-relational models that the view is subscribed to change events of by the library. String search is used to avoid requiring a JavaScript parser on the frontend or a build step when developing, but it may be difficult to parse conditional assignments of model attributes or relations to variables, which can most likely just be not optimized. The library intelligently merges in models and attributes referenced by child components, which are called "fragments", into one query definition for each data loading component. The library also maintains an optional cache of all previously fetched models that correctly handles partial row data.

An alternative to Facebook's Relay[23], which uses GraphQL, a text format for queries (data requirements) with no model layer.

[22] `http://github.com/mwhite/react-jsonapi`
[23] `http://relay.dev`

4.3 Elimination of deploy-time database migrations

In traditional web applications that use a relational database, database migrations are required to handle changes to the structure of the database. *Schema migrations* are created by the web framework when table or column definitions are added, changed, or removed. *Data migrations* can also be created by the developer when stored data needs to be recalculated to work with a change in the application. Migrations traditionally cause downtime due to the need to wait until they finish to run the new version of the application and can cause more significant downtime if a migration fails, though unlikely. Some NoSQL databases require migrations to be performed in the application code as data is accessed. The framework solves the problem of downtime from database migrations without NoSQL by elimination of deploy-time migrations with one method for eliminating schema migrations and one method for eliminating data migrations.

Schema migrations are eliminated by storing all data in a JSON column[24] and creating SQL views on each deploy for the latest schema version (or potentially all schema versions that have an application actively using them) to emulate a standard table, with default values based on schema version id (a sequential integer stored alongside the data) that default to NULL and can be changed as part of the column definition in the application source, and triggers to emulate foreign key constraints. This can be implemented as either a standalone database extension with DDL (Data Definition Language) query rewriting (specify a column named e.g. json_table with schema history and defaults as a JSON string `DEFAULT` or specify an SQL comment with a proprietary delineator as configuration to avoid the need for any changes to the SQL standard) or with application-layer logic in the backend deployment part of the framework. `CREATE TABLE`, `CREATE INDEX`, `CREATE VIEW`, and `CREATE TRIGGER` statements would be rewritten by the extension in terms of the current schema of the JSON column if the extension is offered as a standalone product outside of the framework. Any web framework can be modified to ignore migrations generated by the framework and instead issue the full suite of CREATE statements at every deploy, triggering recreation of views for the current schema. The raw underlying JSON column should be made available so past data can be accessed even if it is not represented in the current schema, so column changes are not required to be add only.

Deploy-time data migrations are eliminated through query-time execution of algebraic data migrations in an ORM extension, with optional incremental background

[24]JSON Types - PostgreSQL. `https://www.postgresql.org/docs/current/datatype-json.html`

execution, with the possibility of temporary multiple-version unmigrated data for very large databases or expensive migrations, to keep the application always using only the latest version of the schema, unlike applications written with some NoSQL databases. This can be implemented by defining a function that returns all the rows needed to calculate the new version of the data for one or more rows having an incremental data migration performed on it, and then updating each row (again with a schema version ID stored alongside the data) using the ORM before executing the original query.

4.4 Automatic management of database indexes from inspected application queries

In most applications, database indexes are necessary to achieve the desired level of performance. Databases provide several types of indexes that are usually managed manually by the developer as part of the application definition, with indexes being created and deleted as part of deployment.

Instead, the framework can include automatic index management on deployment using the online index creation capability of modern relational databases with zero downtime based on inspected application queries, statistical samples of production data, and annotated view and API endpoint performance requirements. This can be achieved by straightforward adaptations of the query planner. Specifically, the function that returns whether an index exists can be replaced with a different version that is used to test all possible indexes that would be used for a given query and all possible permutations of index availability for all queries. Another approach is to implement the index management as a database DDL function that either looks up the application queries from tables they are stored in by the framework or takes queries as a SELECT statement argument that can be populated explicitly by the framework. An optimization problem must be solved involving required or ideal view performance and index size and performance across all views.

All queries that are not significantly programmatically constructed (i.e., in a loop) can be parsed from the application source, with variables referenced in the query definition code being analysed in terms of which model attributes they reference, which may have a known set of values from real data, or more in-depth analysis for custom endpoints. It may be possible to integrate programmatically constructed queries with deeper code analysis.

Explicit annotated view and API endpoint performance expectations in the source code is considered preferable to automatic management based on monitoring real-world usage, but the latter is also possible.

4.5 Full-stack pseudo-transactions with a conflict resolution API

A JSON API REST API with Backbone-relational models represents the fundamental model complexity of a database-backed application frontend. However, applications typically require transactions when the user makes modifications to multiple models before clicking a save button, which in a web application would usually have to be implemented using a custom API endpoint or avoided by requiring each model to be saved before another model is edited.

Instead, if a variant of versioned domain model is enabled (see 4.8.1), a generic multi-model patch or put (similar to HTTP PUT) method can be provided that takes a list of model changes and previous version IDs and returns success with new version IDs if all the model changes can be applied without conflicts in a single transaction or a list of conflicts with data that has been added since the previous version ID, enabling the developer to provide a conflict resolution UI, with a default one provided by the library. Changes need to be saved as soon as a model edit view is switched away from in order to avoid losing data, but this can be accomplished by having an ID for a pending change on the backend be returned and stored on the frontend in order to be passed to the multi-model patch method when a save button is pressed.

This can be implemented using the backbone.trackit [25] change tracking library for Backbone and by checking for conflicts within a standard transaction on the backend.

The same library should also handle updating models that have been loaded on the frontend with new changes from the backend, such as from other users, and notifying the user, if needed.

Note that transactions would initially appear to only work for operations on individual models, not bulk queries. However, bulk queries can be integrated into full-stack pseudo-transactions by maintaining a latest accepted server changes timestamp for each active transaction and having the backend query return an additional value indicating whether the query would return a different result for rows that have been updated by other completed transactions since the timestamp and providing an API to return the new result and update the timestamp to the current time. Note that this does not fully implement true isolated transactions because each bulk query can only return the latest data, and a transaction can complete that would conflict with a transaction opened before it if it saw the latest data at a different time. However, a conflict resolution API that integrates bulk queries can be offered. The transaction

[25] backbone.trackit

logic can be extended to implement nested transactions.

4.6 A SOLID plugin system and registry that supports model ontologies

To enable fully modular application definition, a plugin system that encapsulates both frontend and backend functionality in reusable plugins is necessary. For the frontend, all functionality can be defined as plugins that call each other through common method names without knowing what order they are installed in, while for the backend a simpler plugin system with explicit dependencies of custom endpoints on plugin methods is likely sufficient.

- Database query "scopes" (model class methods that take a query and one or more other arguments and return a new query modified based on those arguments) defined per model for reusable query narrowings.
- Extensible[26]: A plugin library for JavaScript that wraps each plugin method by extending the passed prototype with a method that calls the method of the same name on the most recently installed plugin to provide definition of plugins that do not know anything about other plugins that implement the same methods and can be installed in any order, requiring the developer to intelligently segment plugin functionality, as seen in the JSTree JavaScript tree view component project.[27].
- Plugin capability using Extensible for the top-level application component, with a method under plugging for registering plugins on other view components or similar and a method that returns available components and their capabilities. In addition, a system for declaring capabilities and dependencies for ORM models and query scopes that reuse a global ontology of model and column roles so that plugins can reuse the same data models without column names being hard-coded.

A view component and plugin registry can be developed as a configurable companion to the editor. For applications using web views, the optional common widget library (see 3) can be provided through the plugin system.

[26] `https://github.com/mwhite/extensible`
[27] JSTree

4.7 Automatic application translation at deploy time

Traditionally, applications are manually translated using utterances translated for each language defined with the application by translators. Instead, it is possible to implement automatic translation of application text at deploy-time using a combinatory categorial grammar[28] (a linguistic parsing formalism that uses equational proofs of utterances from a lexicon specified in terms of a part of speech missing a part of speech to either side) library with a semantics module with a lexicon requested on-demand with caching from a free multi-lingual online dictionary (Wiktionary) with navigation and help text used for heuristic disambiguation or full semantic constraint satisfaction disambiguation. This slightly increases deploy time, primarily at first use.

Wiktionary has definitions for words in all languages available in any language, with the definition language edition size ratio similar to Wikipedia edition sizes, with a subset of words having explicit translations available. It is believed that an algorithm exists for exact translations that disambiguate words with multiple definitions based on this data involving recursive constraint satisfaction on the definition contents of any defined words or phrases in an utterance and can be integrated into a CCG library.

While desirable on its own, automatic translation can be considered required for an editor that can edit any application due to applications in theory differing between languages. In addition, sourcing the lexicon from a collaborative online dictionary in theory allows for any rate of language change.

4.8 Description of the framework

Additional components added to the 8-layer declarative object-oriented architecture:

- Replacement of the Python backend web framework with a Deno[29] (the successor to Node.js[30]) JavaScript framework that features a JavaScript API for declarative definition of SQLAlchemy ORM models and queries with Python code made available in JavaScript through a bridge[31] and manual streaming of the Python REST API generator response, with transpilation for ORM operator overloading. It is considered desirable to continue enabling the user to write backend code in Python if desired for its greater expressiveness.

[28] `http://en.wikipedia.org/wiki/Combinatory_categorial_grammar`
[29] `https://deno.com/`
[30] `http://nodejs.org`
[31] `https://github.com/denosaurs/deno_python`

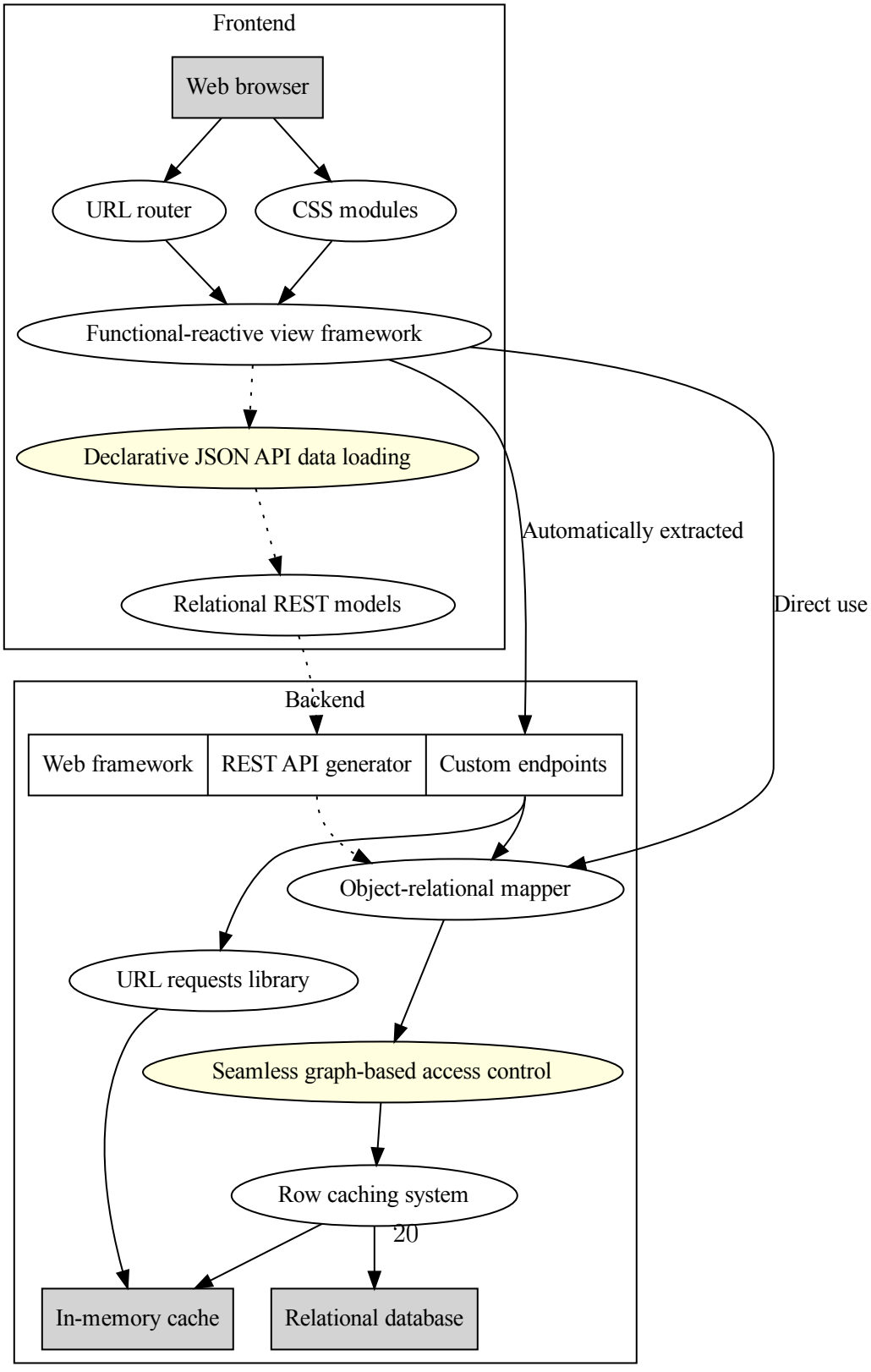
  - Seamless integration of the ORM and the functional-reactive view framework with render methods able to be written as if ORM queries are executed synchronously in the render method of a view component through automatic extraction of custom API endpoints from frontend view code, in addition to optional explicitly defined custom endpoints, by parsing the view code and automatic use of the REST API generated by the REST API generator for one-query transactions. Automatic creation of Backbone-Relational models from the ORM models.
  - All custom endpoints automatically altered to return a row ID for any row referenced individually or in the result of a join so all data is stored in Backbone-relational models on the frontend with deduplication and granular caching using the same system as React JSON API (see 4.2) except for queries that return an aggregate function at the top level. Aggregates across joins stored as a special Backbone model attribute with integration with transactions (see 4.5).
  - Traditional server-side rendered HTML REST output not allowed.

- All additional data processing is required to be implemented using ORM instance events in JavaScript that get run when either the REST API generator or a custom endpoint is called, defining event handlers using the bridge in a minimal transpiled functional embedded language (a LISP, most likely) to enable point-and-click editing (no database triggers allowed in order to enforce privileges in the ORM)

The framework can optionally be implemented in LISP or a domain-specific language (DSL) that compiles to JavaScript with the keywords being Model (IsMultiTenant (see 4.8.1), IsVersioned (see 4.8.1), IsTimestamped (see 4.8.1), Privilege, Field (Privilege), HasOne, HasMany, Property, Constraint, Aggregate (see 4.8.1)), Task, Component (ShouldUpdate), Route, Endpoint, and Plugin. Explicit database index and frontend data requirement definition should never be required but can potentially be included as an option.

4.8.1 Additional features

A number of additional features are required to be able to implement standard web applications using the framework. User-facing features can be implemented as plugins using the SOLID plugin system (see 4.6).


Frontend
Web browser
URL router
CSS modules
Functional-reactive view framework
Declarative JSON API data loading
Relational REST models
Automatically extracted
Direct use
Backend
Web framework
REST API generator
Custom endpoints
Object-relational mapper
URL requests library
Seamless graph-based access control
Row caching system
In-memory cache
Relational database

- User management and authentication. Current user is set as an SQLAlchemy session variable for privilege calculation. HTTP Digest Authentication[32]. Login, register, and reset password functionality.
- Row-based multi-tenancy and soft delete implemented through subclasses of the SQLAlchemy GBAC PrivilegedMixin class with automatic definition of a tenant ID column for top-level multi-tenant models and prepending to index and constraint definitions.
- Optional created_at and updated_at timestamp columns.
- Only allow auto-incremented integer primary keys by default, but with unique constraints available for use with columns that might be a natural primary key in some hand-coded applications. Setups with a multi-master database or other horizontal scaling methods that require it may use UUID primary keys.
- Optional manual translation with a web UI or configuration files
- Server-side rendering[33] of the initial functional reactive view state (built-in to React) *Implementation note:* "Suspense"[34] functionality for partial server-side rendering considered unnecessary, preferring to just serve the initial view with no data if data cannot be fetched quickly enough. "Server components"[35] considered out of scope.
- Background tasks with the database as the queue storage with an admin UI for monitoring
- Templated SMTP email with an admin UI for sent emails and manual emails.
- Upload of non-code static assets such as images with integration with a Content Delivery Network (CDN).
- Per-account feature flags that enable a specific feature, as hybrid properties on an AccountFeatureFlags model that reference manually configured flag values if enabled or an attribute of the active plan of the billing plugin (see below) if installed, with the available flags defined in a configuration file. Referenced in privilege definitions of any model through the current user. UIs can either query

[32] `https://en.wikipedia.org/wiki/Digest_access_authentication`
[33] `https://legacy.reactjs.org/docs/react-dom-server.html`
[34] `https://react.dev/reference/react/Suspense`
[35] `https://react.dev/reference/rsc/server-components`

whether a feature flag is set or send a no-effect test request to the operations they use to see whether they have the required privileges and display a different UI if not.

- Integrated API analytics and frontend analytics, the latter by instrumenting click handlers. Set visitor data as SQLAlchemy session variables for declarative privilege calculation based on IP address or referrer. auditing can use the same infrastructure as analytics.

- WebSockets[36] integration for bi-directional real-time updates of models using Flask-SocketIO[37] and Backbone.iobind[38]

- Business intelligence: automatic generation of Mondrian[39] schemas for base ORM models without an Extract-Transform-Load (ETL) step, with aggregates defined alongside the ORM models with optional automatic materialized views, with an integrated BI UI[40]

- Billing for commercial applications with one-time and recurring payments by abstracting over a single Stripe[41] API account. Provide support for subscription plans that implement feature flags, configuration variables, and numeric functionality quotas.

- Versioned domain model as an ORM mixin for non end-user data versioning (SQLAlchemy Continuum[42])

- Functionality for writing and running functional and unit tests of frontend code and custom API endpoints.

- Backend API versioning.

4.8.2 Performance optimization

Several optimizations enable virtually unlimited scaling when using a relational database with no or low involvement by the application developer:

[36] WebSockets. `https://en.wikipedia.org/wiki/WebSocket`. Wikipedia.
[37] `https://flask-socketio.readthedocs.io/en/latest/`
[38] `https://github.com/noveogroup/backbone.iobind`
[39] `https://en.wikipedia.org/wiki/Mondrian_OLAP_server`
[40] `https://github.com/OSBI/saiku`
[41] `http://stripe.com`
[42] `https://github.com/sqlalchemy-continuum/sqlalchemy-continuum`

- Optional explicit caching of full and partial database rows in Memcached[43], an in-memory cache. `RETURNING`[44] must be used to return values modified by a query on all `UPDATE` queries in order to update the cache.
- Automatic database index management (see 4.4)
- Automatic creation of appropriate denormalized aggregate columns from inspected application queries using SQLAlchemy-Utils' aggregate column functionality that automatically updates aggregate columns on relevant changes, and runtime expression substitution of them in ORM queries
- Automatic SQL `TABLESPACE`[45]-based database partitioning for multi-tenant applications by sharding based on tenant ID, with automatic repartitioning and modification of the sharding function based on data usage and storage availability changes for any disk drive device that is marked as a database storage device.
- CitusDB[46] or multi-master horizontal scaling extension for PostgreSQL available for huge amounts of data not conducive to vertically scaled row-based multi-tenancy.
- Optional caching of external URL requests marked as idempotent and custom endpoint responses in Memcached.

5 A Self-Hosting Point-and-Click Editor for Any Functionally-Specified GUI-DB Application

5.1 Data duplication in version control by the ORM as an alternative to versioned domain model for user-edited applications

- SQLAlchemy Git[47]: To enable editing of the application with branching version control rather than linear version control, declarative ORM model mixins that

[43] `http://memcached.org`
[44] `https://www.postgresql.org/docs/current/dml-returning.html`
[45] `https://www.postgresql.org/docs/current/manage-ag-tablespaces.html`
[46] `http://citusdata.com`
[47] `http://gitlab.com/mawhite/sqlalchemy-git` (private repo, incomplete)

make the model definitions support versioning while remaining declarative are developed, specifically `RepositoryMixin`, `BranchMixin` and `PerBranchMixin` ORM models that allow the current state of each branch to be duplicated between the database and JSON files in the Git[48] branching version control system by traversing models outward from those inheriting from `PerBranchMixin` and creating a nested JSON entry for each row in one JSON file per table, using a native Git library with ORM extension hooks. Read-only access to previous revisions from Git can be provided using the same API as the ORM. Used for user-facing version control instead of versioned domain model to enable manipulating version controlled data through the REST API without having to deal with data de-duplication across versions in a versioned domain model, and so external applications can use the Git repository directly (implement a Git post-receive hook[49] that updates the database). *Implementation note:* Always store the current working state of each branch as a "stash commit" to enable switching between branches in Git's working copy to edit a new branch.

5.2 Description of the editor

A standard point-and-click functional-reactive frontend component editor can be extended to algorithmically display the outcomes of all relevant use cases based on database fixtures or forked real data during editing to cover all possible use cases. Frontend model property references are auto-completed from the data model. The backend can have the ORM model definitions and custom endpoint queries stored in the database, with privilege definitions and hybrid properties by nature being essentially fully declarative and custom backend APIs being constructed using a minimal transpiled functional language.

- Point-and-click/click-and-drag editing of functional reactive view components with data fixtures or forked real data for testing and structured editing of conditional expressions embedded in views, with event handlers written in a minimal transpiled functional embedded language that exposes a JavaScript functional array utility belt library (Underscore[50]) to enable point-and-click editing of event handlers.

- A self-hosting application code editor that parses and stores ORM model and view component definitions in the database, the latter as a high-level abstract

[48] `http://git-scm.com`
[49] `https://git-scm.com/docs/githooks`
[50] `http://underscorejs.org`

syntax tree, with standard IDE features for any extraneous parts of the application that require imperative/custom code to ensure compatibility with the JavaScript ecosystem

In order to have the editor able to edit itself effectively, version control is required, which can be a frontend view that displays structured descriptions of changes to the application structure, similar to GitHub.com, with merge requests that include a conflict resolution tool.

Every committed version of the frontend can be made available through a unique URL. Minification and bundling of the latest frontend version (though the latter is typically considered obsolete with HTTP2, but may still be necessary for applications with a large number of files including dependencies), with intelligent separate bundles for shared dependencies, can occur immediately after restarting of the latest backend version, with the previous frontend version that may display a helpful error for features that require an earlier version of the API being served until it is complete, leading to no-moving-parts deployments that require only restarting the web application server when used in combination with the elimination of database migrations (see 4.3).

List of Software

5.3 Proposed products

- AppAlchemist (working name) - high-level JavaScript web framework
- AppAlchemist IDE (working name) - point-and-click editor for AppAlchemist

5.4 Core programs depended on

- JavaScript programming language
- Chromium browser/V8 JavaScript engine
- Deno JavaScript runtime (the successor to Node.js)
- Python programming language
- PostgreSQL relational database
- Memcached in-memory cache

- Mondrian OLAP business intelligence server
- Git version control system

5.5 Libraries

Name	Description	Author	Date	URL
Flask	Python web micro-framework	Armin Ronacher	2010	`http://flask.palletsproject.com`
SQLAlchemy	Python object-relational mapper	Michael Bayer	2004	`http://sqlalchemy.org`
Flask-SQLAlchemy	Flask/SQLAlchemy integration			`http://flask-sqlalchemy.readthedocs.io`
Flask-Restless NG	REST API generator	Jeffrey Finkelstein		`http://flask-restless-ng.readthedocs.io`
SQLAlchemy-Continuum	SQLAlchemy versioned domain model			
SQLAlchemy GBAC	Graph-based access control for SQLAlchemy	Michael White	In progress	
SQLAlchemy Utils	Aggregate columns for SQLAlchemy			
Backbone	JavaScript REST Models	Jeremy Ashkenas		
Backbone-Relational	Relational models for Backbone	Paul		
React	JavaScript functional-reactive views	Facebook, Inc.		
React Router	Frontend URL router			
React JSON API	Declarative Backbone-relational JSON API data loading for React	Michael White	2017	

React Intl	Internationalization for React	NFL		
Formik	React forms			
Babel	JS transpiler			
Rollup	JS bundler			
Babel-plugin-react-css-modules	CSS Modules for React			
Underscore	JavaScript functional array utility belt	Jeremy Ashkenas		
pg_jsonview	JSON tables with views and triggers for PostgreSQL	Michael White	TBD	
Structor	React component editor			
Extensible	Decoupled nesting plugins for JavaScript	Michael White	c. 2016	
SQLAlchemy Git	Versioning in Git for SQL-Alchemy	Michael White	In progress	
Deno_python	JavaScript to Python bridge			
Flask-SocketIO	WebSockets for Flask			
backbone.iobind	WebSockets integration for Backbone			
pg_autoindex	Automatic index management from stored query definitions for PostgreSQL	Michael White	TBD	
Flask Aggregate Columns	Automatic aggregate columns for Flask	Michael White	TBD	

Glossary

abstract syntax tree A high level representation of code after it has been parsed and lexed (converted into tokens).. 3, 24

aggregate column A database column that stores a value calculated for each row in a table as an aggregate over rows in another table related by a foreign key or a subset of related rows. 23

algebraic Characterized by a formal definition of inputs and outputs in an expression language. 14

analytics Tracking of usage data for an application. Frontend analytics track the sequence of events while viewing a page. 22

API An Application Programming Interface, the set of methods exposed by an application that clients can call. 4, 29, 31

auditing Logging of access to pages. 22

backend A part of an application that is not exposed to the end-user. Typically interacts with a database and possibly external services. 4, 31

bridge A modification to a programming language's runtime that enables it to call functions written in another programming language while sharing the same memory and call stack. 3, 18

bundling Combination of multiple frontend source files into one. 25

business intelligence Data analysis tools for calculating and visualizing the value of aggregations of a company's data over any time period. 22

closure A function that encapsulates the calling environment (i.e. variables) that can be stored as a variable.. 31

Content Delivery Network A website that provides a single URL to download a frontend asset or bundle from one of many servers geographically close to the user. 21

CRUD Create, Read, Update, Delete. The types of operations on rows in a relational database. 10, 33

CSS Cascading Style Sheets. 5

data migration A recalculation of data stored in the database for some or all rows. Executed in the application layer. 14, 29

database fixture Test data for an application that is intended to cover possible use cases. 24

database index An in-memory ordered data structure created to increase the speed of database queries that are expensive to perform using disk reads. 15

database migration A change to a deployed application's database structure or data traditionally executed at deploy-time. Can be either a schema migration or a data migration. 14

database partitioning Partitioning of a database into disjoint subsets of the data with storage on different devices or servers. 23

database trigger A function defined by the developer that is run by the database when a row in a table is added, changed, or deleted within the existing transaction in order to make additional changes to data. 19

denormalized Characterized by duplicate storing of data that can be calculated from other data, i.e. violating normalization. 23

deploy The uploading of code to a production server and restarting of the web application server process with new code. 14

DOM Document Object Model, the in-memory data structure representing the elements on an HTML page. 31, 34

endpoint A method of an API, especially an HTTP API. Often referred to as an API. 4, 32

extension hook A method that can be implemented in an extension to modify the behavior of an extended library. 4

Extract-Transform-Load A step in traditional business intelligence architectures that stores values calculated from a source database or table in a new database or table. 22

feature flag A conditional variable stored as part of an application's configuration or in the database that is referenced in application code to enable or disable specific features implemented in specific code paths. 21

frontend The externally exposed or user-facing part of an application. Typically refers to the code that executes in a web browser for web applications. 4, 5, 31

functional A type of programming language in which code is normally written without making use of side-effects. Also known as declarative. 19, 31

functional-reactive A type of user interface rendering library in which the UI is constructed as a tree of components whose rendered output is each a pure function of properties passed to the component and state maintained internally by the component. Web-based functional-reactive view libraries typically use a virtual DOM (Document Object Model) to make display changes more efficient.. 2, 31

graph-based access control A type of access control in which privileges are expressed as a function of the entities and their properties in the entity graph of an application's data model. Any more specific access control formalism, such as role-based access control, can be expressed using graph-based access control by modeling the access control data naturally. 2, 10

horizontal scaling Scaling to higher usage by adding servers, in contrast to increasing the amount of resources per server. Requires solving coordination problems. 4, 23

IDE A text editor with code understanding functionality specific to the programming language. 3, 25

idempotent A property of a function indicating that it has no effect after the first time it is called with a given set of arguments. 23

inspected Information parsed from application source. Can also refer to a unified API for getting information about an object in an application. 15

instrumenting Recording each time an event handler is called, e.g. by wrapping a method with a new method. 22

JavaScript An interpreted object-oriented language with prototype-based inheritance and closures primarily used as the scripting language in web browsers, where it has an API for the DOM and asychronous HTTP requests, but also used on the backend created by Brendan Eich in 1995. 4, 5

JSON API A REST API standard based on JSON (JavaScript Object Notation) with embedded related entities, filtering, and pagination.[51]. 4

lifecycle method In a functional-reactive view framework, a method on a component class that the framework calls to perform some operation, such as responding to initial mount or changes or short-circuiting rendering based on some calculation from data. 5

LISP A functional programming language released in 1962.. 19

materialized view An SQL view that caches and periodically updates, typically once a day, the rows in the view, to avoid performing expensive calculations every time the view is referenced. 22

minification Reduction of frontend source file size by replacing applicable identifiers with shortened names and eliminating dead code. 25

modular Decomposed into modules. 17

multi-layer Organized into multiple layers that each provide a different part of the functionality and call each other. Most commonly used to refer to frontend and backend layers.. 2

nested transaction A sub-transaction within a relational database transaction that can be rolled back to a savepoint without rolling back the parent transaction. 11

NoSQL A trend in the early 2010s of databases that achieved horizontal scalability by dropping strong ACID (Atomic, Consistent, Isolated, Durable) guarantees, requiring the application to handle exceptional cases itself, or exposed a query model that enabled high scalability with less powerful queries, such as MongoDB[52] and CouchDB[53] [54]. 4, 10, 14

[51] `http://jsonapi.org`
[52] `http://mongodb.com`
[53] `http://couchdb.apache.org`
[54] `http://en.wikipedia.org/wiki/NoSQL`

object-relational mapper A database access library that provides an abstraction for constructing relational database queries using the application programming language instead of SQL and loading query result rows into objects. The database tables are defined as classes with class attributes for columns, typically corresponding to the full range of SQL DDL (Data Definition Language). Typically provides functionality for including desired related entities in the query and automatically loading them into nested objects with duplicate entities in the results referring to the same object. Integrates with database transactions. 2, 4, 33, 34

operator overloading A programming language feature that calls a custom implementation of standard mathematical operators based on the type of the argument. 18, 32

prototype An object attached to JavaScript functions that determines the base object inherited from when a function is called as a constructor. Can have its own prototype property for reuse. 17, 31

Python A high-level interpreted object-oriented language with metaclasses, dynamic attribute lookup, operator overloading, decorators, iterators, generators, fully qualified dynamic imports, dictionaries, list comprehensions, and one-expression anonymous functions created by Guido van Rossum in 1989. 2, 4

query planner The part of a relational database that selects which types of scans of data on disk to perform and which indexes to use in order to execute a query. 15

relational database A database consisting of tables with rows and columns and relations defined on those tables in terms of foreign keys. Queried using SQL. Supports ACID (Atomic, Consistent, Isolated, Durable) transactions. 4, 10, 28, 31–33

REST Representational State Transfer, the name for HTTP APIs that use standard HTTP methods, GET, POST, PUT, and sometimes DELETE, usually on endpoints that represent individual database rows or collections. 4, 31

schema migration A database migration in which tables, columns, constraints, views, or triggers are added or deleted. Traditionally leads to downtime for

addition of new columns due to the need to rewrite data on disk with the new column. 29

seamless A property of code that does not have any seams, i.e. it perfectly encapsulates functionality within an abstraction. 2, 10

self-hosting Able to compile itself. Typically used in reference to programming language compilers. 3, 24

sharding The separation of rows in a database into disjoint sets for separate storage, often based on a function of the primary key (ID) of the row. 23

single and multi-table inheritance In a database schema, the storing of a class hierarchy in the database by storing the class of each row as a column. Multi-table inheritance is duplication of rows in one table in one of multiple other tables depending on the class that "inherits" from the first table using a foreign key reference on a primary key to the first table's primary key and stores additional data for only that table. Single-table inheritance is the storing of the same data for all classes in one table. 4

SOLID A set of software design principles for object-oriented code. Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion. 3

SQL Structured Query Language. The query language for relational databases. Allows CRUD operations with joins of related tables, aggregation, and built-in data processing functions. 4, 32, 33

SQL view An SQL query that can be referenced like a built-in table. 14, 31

template engine A library that renders an HTML string from a template including references to variables passed as the "context" and conditional logic or loops encoded using tags. 2, 34

transpilation Compilation of one high-level language into another that is often a subset of it. Typically used with JavaScript. 18

unit-of-work An object-relational mapper pattern in which every change occurs inside a transaction in a batch, contrasted with "active record", in which model objects directly manipulate live database state by default. 4

URL router A library that renders view handlers registered at specific URLs, usually with dynamic URL components passed to the view. Can be either frontend or backend. 5, 13

UUID Universally Unique Identifier, an ID in which no two generated IDs ever conflict. 21

version control Tracking of changes to a codebase. 3, 25

versioned domain model A method of storing the full change history of all tables in the database by storing a new row with a version ID for each change to the current data. 16, 22, 24

virtual DOM A virtual mirror of the browser's Document Object Model (DOM) maintained by a functional-reactive view framework to improve performance by batching updates and re-using existing DOM nodes when changes occur. 5, 30

web application An application that is run in a web browser. Can have interactive functionality using JavaScript. Typically has a backend server component. 2, 34

web application server A library that receives HTTP requests and generates responses by calling handlers registered by the developer that match a route pattern based on the request URL. 2, 34

web framework A library for creating web applications. Typically has a web application server and template engine. Often has an object-relational mapper. 2, 4