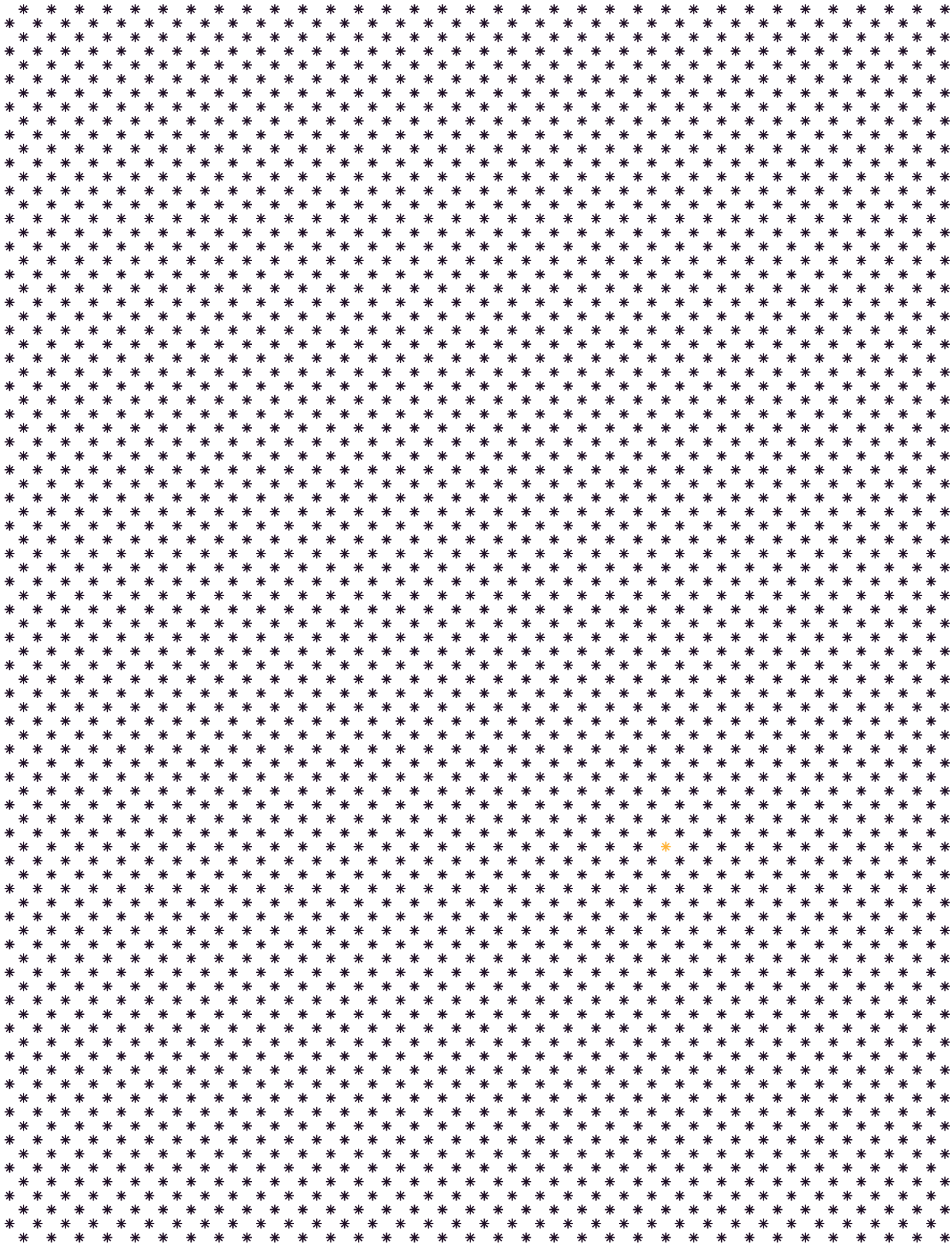

Summary

Starcounter is a full-stack enterprise application platform. It combines in-memory database management system (DBMS), an application server and libraries enabling front-end development. It's built to make applications with high performance, minimal development effort, and low total cost of ownership. Starcounter can be a basis for applications in a multitude of architecture patterns and infrastructure types.

The foundation of Starcounter is the virtual machine database management system (VMDBMS) that keeps data in one place instead of moving it between the application and the database.

The next layer, the application layer, contains API's to use the VMDBMS - such as transactions, queries, and schema definitions. Also, the application layer has tools for interacting with view-models, sessions, and other utilities for building modern server-client applications.

The network layer is built on top of the application layer and provides real-time client-server communication with as low latency as possible. It supports network protocols such as HTTP, WebSocket, and TCP.

Front-end libraries integrated with Starcounter enable adhering any modern web framework with performance, simplicity, and real-time user experience in mind.

All layers can be used independently from each other due to a modular architecture. Starcounter can be used as a very performant in-memory database, application backend, or full-stack platform to build fully-capable web applications.

Introduction.

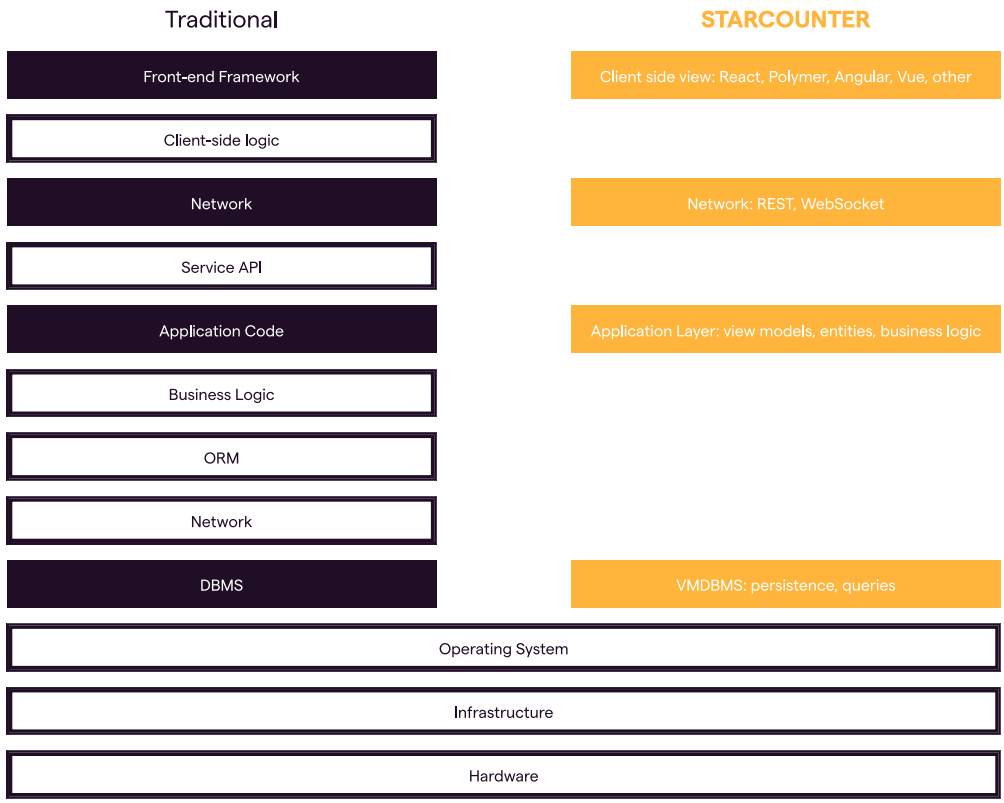

Figure 0: Overview of the Starcounter architecture

Scope

This document focuses on Starcounter's architecture - the different components and how they are connected. It does not cover how to build Starcounter apps, tooling, security, deployment and similar aspects. This version of the document describes Starcounter as it works in version 2.3.2.

Audience

This document is for technical readers that want to understand Starcounter. This includes, but is not limited to, developers that want to get a background before starting development, Chief Technology Officers, software architects and others that decide about technology platforms, and existing users of Starcounter that want to improve their understanding of Starcounter fundamentals.

Prerequisite Knowledge

To understand everything in this document, you should know these technologies and concepts on a fundamental level:

- Database concepts such as transactions, ACID, and SQL
- Object-oriented programming
- The communication protocols WebSocket and Hypertext Transfer Protocol (HTTP)
- Web technologies including basic understanding of HTML templating and template binding.

The document uses C# for code examples. The examples are simple and are meant to be understandable without knowing C#.

VMDBMS.

Virtual Machine Database Management System.

The Virtual Machine Database Management System (VMDBMS) is a row-based relational in-memory database system based on United States Patent: 8856092 granted by Starcounter. In short, the VMDBMS works like a persistent heap for a virtual machine based language, such as languages built on the .NET Common Language Runtime (CLR) or the Java Virtual Machine (JVM). It currently works with C#. When an app starts, Starcounter allocates a piece of memory to the VMDBMS. Apps run inside the memory of the VMDBMS. With the apps in the VMDBMS - the VMDBMS can store objects from these apps persistently; these objects are called persistent objects. The app can access persistent objects stored in the database. It can also modify these persistent objects like any other objects.

Weaver

The developer defines which classes should have their instances stored persistently by decorating the classes with a particular attribute. We refer to these classes with persistent instances as database classes. The weaver, a tool that transforms the application code to work with the VMDBMS, locates the database classes during compile time. The weaver then automatically rewrites the class to call into the database engine instead of the heap. Thus, the only thing stored in the object on the heap is the identification of a persistent object in the VMDBMS that holds the data. For the developer, it looks similar to the code-first approach in object-relational mappers such as Entity Framework where the developer defines C# classes, and the framework derives a schema from it.

This is what a class looks like before and after weaving:

```
[Database]
public class Person
{
    public string Name { get; set; }
}
```

Figure 1: Database class - before weaving.

```
[Database]
public class Person
{
    private ulong _id;
    public string Name
    {
        get => DbState.GetString(_id, "Name");
        set => DbState.SetString(_id, "Name", value);
    }
}
```

Figure 2: Database class - after weaving.

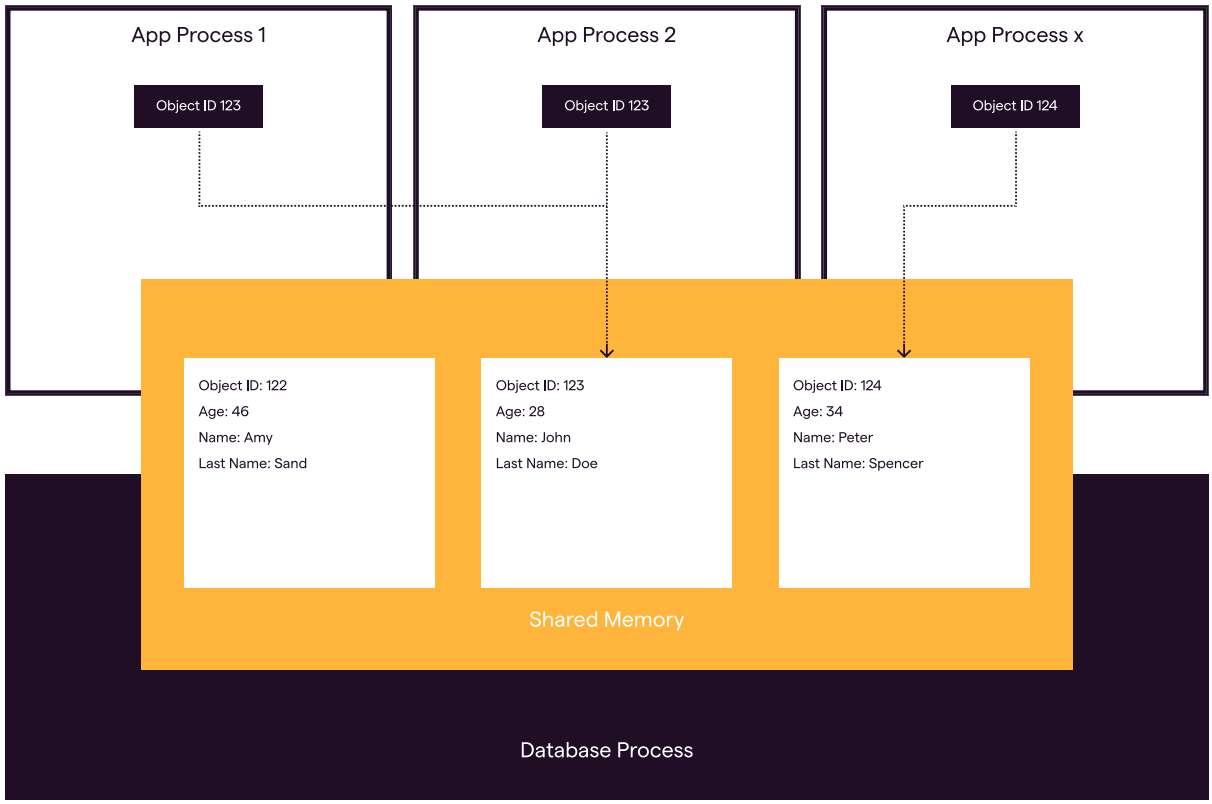

Figure 3: The VMDBMS with multiple apps.

After the weaver weaves the class, whenever the app use the **Name** property, it's retrieving or setting the value in the database and not the object on the heap. With that, persistent objects can be updated the same way as any other object.

Weaved code is not visible to the developer, even if it's used by the compiler. This is by design so that the developer don't have to worry about weaving.

Shared Memory

With the VMDBMS, data doesn't move between the app and the database as in traditional DBMS's. Instead, the VMDBMS modifies the data in place. Keeping data in place increases the performance of data operations, makes the memory footprint smaller and reduces the amount of application code. Since the data object is only stored in one place and the application code only has a reference to that data object, object-relational mappers are not needed.

Apps running in the VMDBMS share the same memory. Thus, all apps immediately see changes to persistent objects. For example, as shown in the figure above, if app process 1 modifies the **Name** property of the object with the identification 123, then all other apps that also hold a reference to that object will see that the **Name** has changed.

ACID

Transactions in Starcounter are atomic, consistent, isolated, and durable (ACID).

Atomicity

Atomicity guarantees that the DBMS either commits all or none of the operations in a transaction. Starcounter ensures atomicity by

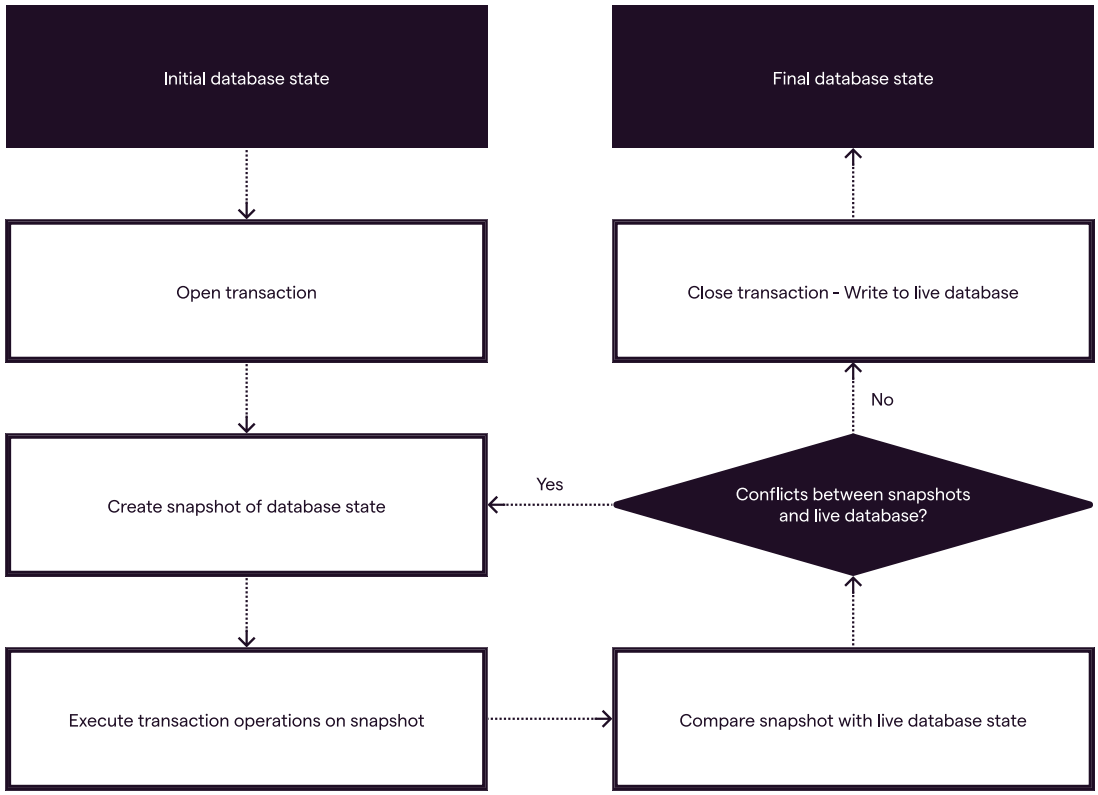

Figure 4: Optimistic concurrency control with snapshot isolation

wrapping changes of one transaction within a transaction scope. The changes commit at the same time. If something interrupts the transaction before the end of the scope is reached, the VMDBMS will not commit any of the changes.

Consistency

A consistent DBMS ensures that all the data written to the database follow the defined constraints of the database. In Starcounter, this is solved by raising exceptions when an action that breaks the constraint is executed in a transaction, such as committing non-unique values to a field that requires unique values. The exception will in turn make the transaction roll back so that none of the changes are applied.

Isolation

To isolate transactions, Starcounter uses snapshot isolation with optimistic concurrency control. With snapshot isolation, all database operations in a transaction are guaranteed to see a consistent snapshot of the database. When the transaction ends, Starcounter checks if there are any conflicts between the live version of the database and the changes made in isolation. If there are no conflicts, the VMDBMS commits the changes, otherwise, the transaction starts over until the transaction is not conflicting with the live database and the VMDBMS commits the changes. Starcounter throws an exception if it takes, by default, more than 100 tries to commit the changes because of conflicts.

Durability

In a durable DBMS, committed changes are persistent, even if there are power failures and similar disturbance. The VMDBMS uses transaction logging to ensure durability. Transaction logging means that database operations are, on commit, written to a log file on a disk. By writing database operations to a log, the VMDBMS can recreate the state of the database by going back in the log.

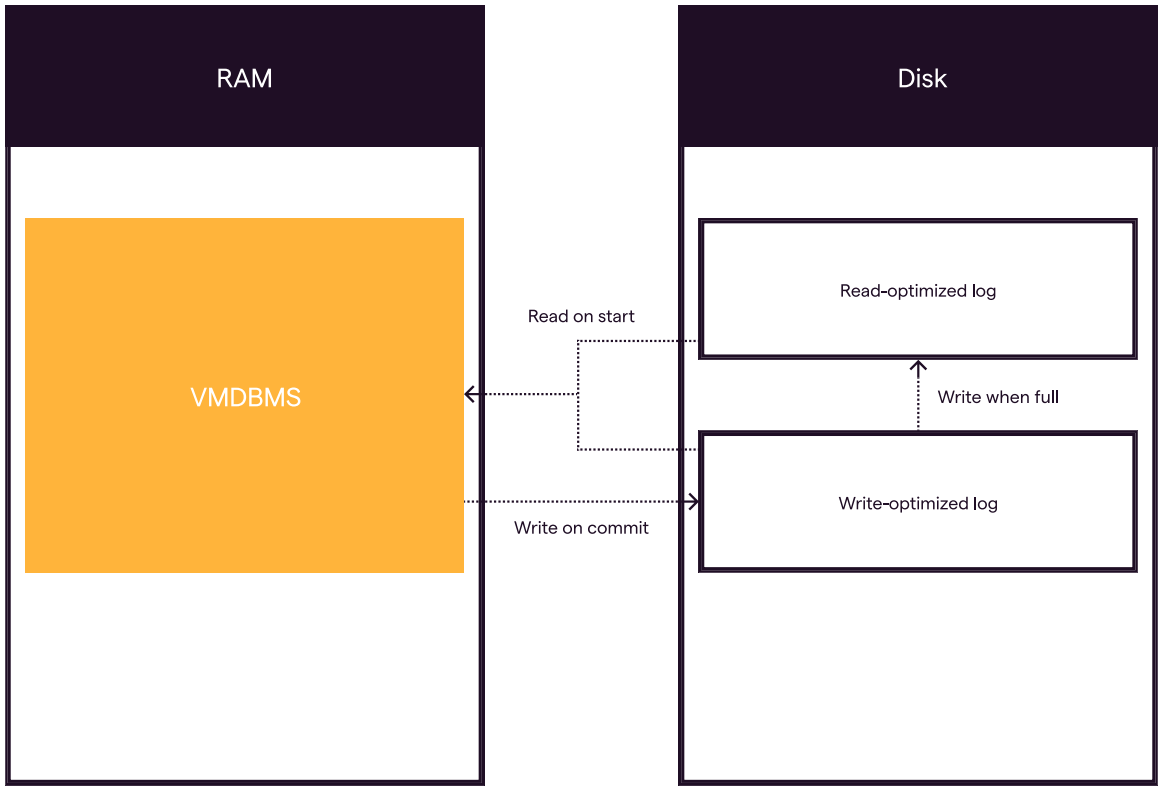

Figure 5: Log file handling

Starcounter uses two types of log files: read-optimized and write-optimized. Write-optimized log files are the files that the VMDBMS writes to when it commits changes. These files are optimized for high throughput. Read-optimized log files are more compact and faster to read from than write-optimized log files. A read-optimized log file is the image of the database at a specific point in time. Read-optimized log files are created from multiple write-optimized log files to save space and make it faster to load data into the VMDBMS.

Asynchronous transactions in Starcounter make the log writes asynchronous. This means that other operations can still be carried out while the VMDBMS is writing to the log.

Asynchronous log writes can make apps faster but also introduce extra complexity.

Queries

Starcounter supports a subset of the ANSI SQL-92 standard. The developer can query the database from both the Starcounter Administrator and from application code. It supports these following statements:

- Data Query Language (DQL): **SELECT**
- Data Definition Language (DDL): **CREATE INDEX, ALTER**, and **DROP**
- Data Manipulation Language (DML): **DELETE**

To better deal with objects, Starcounter SQL supports object extensions according to Object Data Standard ODMG 3.0 (ISBN 1-55860-647- 4). These extensions are object references and path expressions. This often removes the need for complicated and slow joins.

Application Layer

The application layer contains API's to use the VMDBMS, interact with view-models, sessions, and other utilities for building high perfromance applications.
The application layer utilizes the model-view-view-model (MVVM) architecture. The model consists of database classes, as described in the VMDBMS section, the view-models are C# classes called "Typed JSON", as described in this section. The view is a JSON document that can be stateless, to generate a response for a REST API, or stateful to keep an interactive user session for WebSockets.

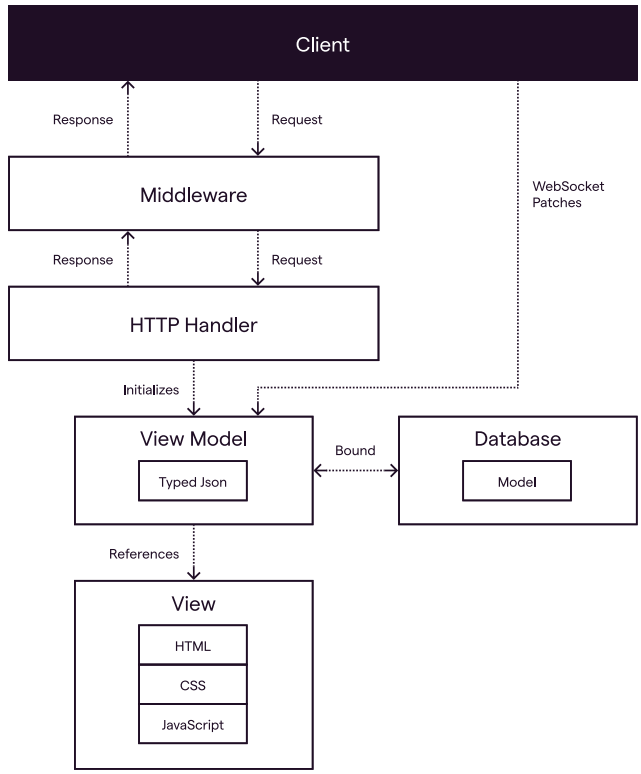

Figure 6: The flow when a client sends a request

Typed JSON

Typed JSON are C# classes that are serializable to JSON. This makes it easy to work with JSON in an object-oriented way. In Starcounter, the view-models are the JSON documents that are created from, and modified with, Typed JSON. View-models are used to define the application logic and manage the application state.
Typed JSON classes are primarily generated by Starcounter from a JSON file that's defined by the developer. The process of generating Typed JSON from JSON is called "JSON-by-example".

With JSON-by-example, the developer creates a JSON tree representing the schema of the view-model, and Starcounter creates a C# class with the same schema as the JSON tree.

This JSON-by-example generates a C# class with the same name as the JSON file with three properties: **Html**, **Name** and **Age**. Name will, by default, be an empty string and **Age** will be zero. The generated Typed JSON class also has some other properties and methods that makes the view-model easier to work with.

A JSON-by-example file that generates a class called **PersonPage** might look like this:

```
{
    "Html": "/People/views/PersonPage.html",
    "Name": "",
    "Age": 0
}
```

Figure 7: A simple JSON-by-example file

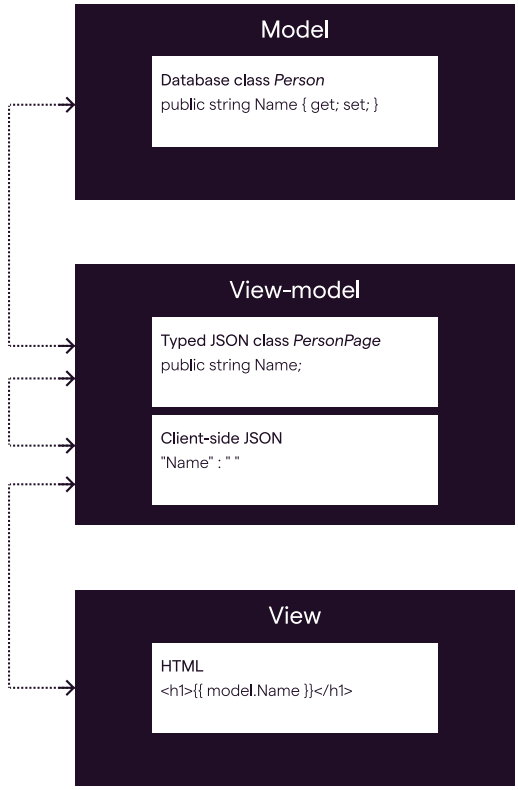

Figure 8: Data bindings between database classes and HTML

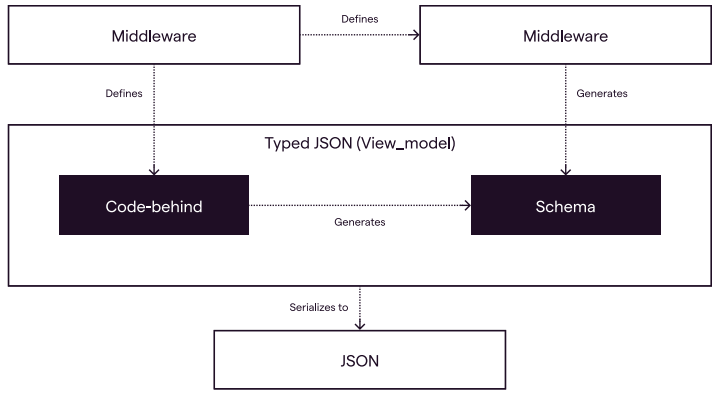

Figure 9: View-model creation with Typed JSON

Properties in Typed JSON can be bound with two-way bindings both to the view and to the model. The bindings to the model makes it possible to have a view-model represent a persistent object - whenever a property in the persistent object changes, the property in the Typed JSON object that's bound to the persistent object changes accordingly, and vice versa. The bindings to the view are explained in the **Front-end** section of this document.

Typed JSON can be extended by creating a code-behind file. This code-behind handles user input and can modify the bindings between the view and view-model. The code- behind is created as a partial class definition of the automatically generated Typed JSON class.

To send the data in a Typed JSON object to the client, return it from an HTTP handler. When a Typed JSON object is returned from an HTTP handler, Starcounter serializes the object to a JSON string: (see Figure 10 below)

When the **/person** URI is called with a **GET** request, the method will execute the lambda that's the second argument to the method which returns a new object of type **PersonPage**. Starcounter will then serialize the returned object to a JSON string. This is also used to bind view-models to sessions.

This allows the developer to create view-models declaratively without serializing or deserializing JSON - that's taken care of by the Typed JSON implementation.

```
Handle.GET("/person", () => new PersonPage()); // =>
{"Name":"","Age":0}
```

Figure 10: JSON serialization from HTTP handler.

Sessions

Sessions are a way to store state that's relevant to individual users, such as view-model state.

The session has an identifier that is used in communication with the client using the available stateless or stateful network APIs (see: **Networking layer**). The session is used by the client to propose the state changes, and by the server to push the state changes. A single client can use multiple sessions, however the common scenario is to use one session per client.

Threading

In Starcounter, all database operations and network operations, such as returning a response from an HTTP handler, have to
be assigned to a thread by a Starcounter scheduler. The scheduler wraps the tasks in
a context that allows them to be executed as database or network operations in Starcounter.

By default, Starcounter creates as many schedulers as there are logical CPU cores.

Network handlers automatically allocate
their tasks with a Starcounter scheduler. Additionally, every session is associated with a specific scheduler. When the developer runs code within a session, Starcounter executes this code on
a thread assigned by the session's associated scheduler.

The number of tasks that can be executed
by Starcounter at the same time is limited,
so to optimize performance, it's good to not schedule tasks with the Starcounter scheduler that don't use Starcounter database nor network features. Good opportunities for such optimization are CPU intensive computations and waiting for asynchronous operations to finish.

Network Layer

Starcounter uses its own web server to handle networking. It supports the network protocols UDP, TCP, HTTP, and WebSocket. The web server sends all requests and responses through the Starcounter network gateway. The developer defines where requests are sent by defining network handlers. The addresses for these handlers are registered in the network gateway that use these to send requests and responses to the right places.

The API's for the network protocols are low level which gives the developer broad control over networking.

Starcounter applications use the networking layer to expose the application state to the client. There is a simple API to build REST-style interfaces. Starcounter server supports an alternative to REST, called Palindrom, which is used to expose the session as a stateful object directly to the client with minimum overhead.

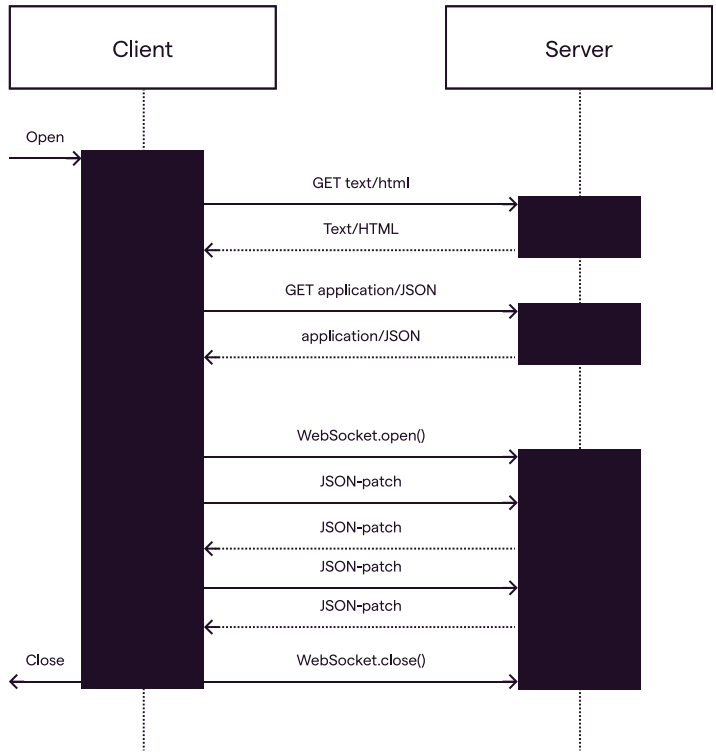

Figure 13: Palindrom protocol

Palindrom

Palindrom is a JavaScript networking protocol and library, researched at Starcounter, that is a low-code alternative to REST-based web services. It is built on JSON Patch (RFC 6902) and WebSocket (RFC 6455). It's used to synchronize JSON trees on the client and the server with as low latency as possible. Two main parts contribute to the reduced latency. First, data sent over WebSocket don't require headers like HTTP requests and responses; this reduces the amount of data sent over the network. Second, with JSON Patch, Palindrom doesn't send the whole application state each time there's a change; instead, it only sends the changes in a format that looks like this:

Palindrom is application-neutral. It supports any programming paradigm that can be implemented using state synchronization between the server and the client. It best suits the use scenarios where the state changes reactively to the user input. Where it really shines, is the ability to implement thin-client applications, which is not possible with REST (see: **Front-end**)

```
[
  { "op": "replace", "path": "/baz", "value": "boo" },
  { "op": "add", "path": "/hello", "value": ["world"] },
  { "op": "remove", "path": "/foo"}
]
```

Figure 12: JSON Patch example.

Front-End

Starcounter supports two kinds of server-client architectures for building GUI applications.

The first kind is building a Starcoutner application with a traditional REST-style API. The client is a separate application that requests the resources from the server. The client application can be developed in any popular technology, such as React or Angular web frameworks or native desktop and mobile SDKs.

The second kind of server-client architecture that is the result of a research in Starcounter is by utilizing the stateful user session. In this architecture, Palindrom is used to synchronize the state object between the server and the client. The user interface can be a thin-client, that is a function of the application state. It is implemented using a reactive pattern with React or a two-way binding pattern with Polymer. The UI can be also a separate thick-client app, that uses Palindrom connection only for communiaction with the server. Starcounter code samples use Polymer, which in our research turned out to be the most effective, declarative way of building user interfaces using the modern Web technology with minimum amount of redundant code.

Starcounter has implemented Palindrom bindings to Polymer JavaScript library and plans to implement bindings for other libraries, based on open sourced definition of the Palindrom protocol.

Conclusion

Starcounter is meant to be used to build applications that are fast, reliable and built in short time. Every component of Starcounter is essential to make this work - from the VMDBMS that brings application to the data to network layer and front-end libraries.

Starcounter is also optimized for developer productivity. Typed JSON, the weaver, and Palindrom all reduce the need for what we call “glue code” - the code used to integrate different components without contributing to the features of the app. This lets the developer focus on the business logic and create value. The developer doesn’t need to configure a database, setup an object- relational mapper, define large REST interfaces, or deserialize and serialize JSON.

Overall, Starcounter vastly simplifies the development and reduces total cost of ownership by using the latest database, network, and web technologies.

Contact

Starcounter AB
Hovslagargatan 3
111 48 Stockholm
Sweden

info@starcounter.com
+46 8-428 425 00

Team

Alexey Moiseenko	Sr Software Developer
Andrej Andrejev	Sr Database Scientist
Daniil Skatov	Head of Development
Erik von Krusenstierna	Software Developer
Erica Larson	CEO
Henrik Sjöström	Query Processor Developer
Kostiantyn Cherniavskyi	Product Owner
Michal Nosek	Product Manager and Tech Sales
Michal Ogrodnik	Sr Software Developer
Per Samuelsson	Sr Software Developer
Vadim Senchukov	Sr Database Scientist
Johan Olsson	Creative Director
Sebastian Otarola	Creative Director
Jacob Holmedahl	CFO
Malin Hurtig	Office Manager