Module-05

MangoDB

MangoDB Basics

MangoDB is a NoSQL database that stores data in a flexible, JSON-like format called document**s** instead of tables. It is designed for **sc**alability**,** high performance, and ease of development.

Core Concepts:

Documents:

The basic unit of data storage in MongoDB, similar to a row in relational databases but in JSON format.

MongoDB is a document-oriented database, meaning that data is stored in document**s** rather than tables. These documents are structured as field-value pairs, similar to JSON objects, allowing for flexibility and scalability.

MongoDB Document Structure:A document in MongoDB consists of fields (keys) and values, where values can be:

- **Primitive types** (strings, numbers, Booleans, dates, timestamps, binary data, etc.)
- **Embedded objects** (nested key-value pairs)
- **Arrays** (lists of values or objects)

For example, an **Invoice document** in MongoDB might look like this:

```
{
  "invoiceNumber": 1234,
  "invoiceDate": ISODate("2018-10-12T05:17:15.737Z"),
  "billingAddress": {
    "name": "Acme Inc.",
    "line1": "106 High Street",
    "city": "New York City",
```

VTUSYNC.IN

```
    "zip": "110001-1234"
  },
  "items": [
    {
      "description": "Compact Fluorescent Lamp",
      "quantity": 4,
      "price": 12.48
    },
    {
      "description": "Whiteboard",
      "quantity": 1,
      "price": 5.44
    }
  ]
}
```

VTUSYNC.IN

Explanation of the Document

1. **Basic Fields**
   - "invoiceNumber": 1234 → Stores an invoice number as a number.
   - "invoiceDate": ISODate("2018-10-12T05:17:15.737Z") → Uses MongoDB's date format for timestamps.
2. **Nested Object (billingAddress)**
   - Instead of storing address details in a separate table (like in SQL), MongoDB stores it **inside the document**.
   - "billingAddress" contains fields like name, line1, city, and zip.
3. **Array of Objects (items)**
   - Instead of using a separate table for invoice items, MongoDB allows an **array of objects** within the document.
   - Each item contains "description", "quantity", and "price".

Advantages of This Approach in MongoDB

Faster Reads – No need for complex joins; all data is retrieved in a single query.
Flexible Schema – No need to define a fixed structure; fields can be added or modified easily.**Efficient Storage** – Stores related data together, reducing redundancy.
Better Scalability – Easily scales horizontally across multiple servers.

Collections:

A group of related documents, similar to tables in SQL databases.

MongoDB Collections and Schema

In MongoDB, a collection is similar to a table in a relational database—it is a group of related documents. However, MongoDB collections offer more flexibility than relational tables.

Key Features of MongoDB Collections

1. **Primary Key (_id Field)**
   - Every document must have a unique _id field.
   - If not provided, MongoDB automatically generates a unique ObjectId for _id.
   - The _id field is automatically indexed for fast lookups.
2. **Indexes for Faster Queries**
   - Apart from _id, indexes can be created on other fields to improve query performance.
   - Indexes can also be created for embedded documents and arrays.
3. **Schema Flexibility**
   - Unlike relational databases, MongoDB does not require a predefined schema.
   - Documents within the same collection can have different fields.
   - However, in practice, most applications follow a consistent document structure.
4. **Schema Validation (Optional, Introduced in MongoDB 3.6)**
   - Allows defining required fields, data types, string lengths, and value rang**es**.
   - Ensures data integrity but is **o**ptional.
   - Errors from schema validation are currently not very detailed.

VTUSYNC.IN

Example: A MongoDB Collection (users)

```
{
  "_id": ObjectId("65a4e6b8c7e9a8d5b1d9a456"),
  "name": "Alice",
```

```
  "email": "alice@example.com",
  "age": 30,
  "address": {
    "street": "123 Main St",
    "city": "New York",
    "zip": "10001"
  }
}
```

Here,

- Id is automatically generated if not provided.
- The document has a nested object (address)**,** showing flexibility.
- If another document in the same collection lacks address, it is still valid.

Databases

- A database is a logical grouping of many collections. Since there are no foreign keys like in a SQL database, the concept of a database is nothing but a logical partitioning namespace.
- Most database operations read or write from a single collection, but $lookup, which is a stage in an aggregation pipeline, is equivalent to a join in SQL databases.
- This stage can combine documents within the same database. Further, taking backups and other administrative tasks work on the database as a unit.
- A database connection is restricted to accessing only one database, so to access multiple databases, multiple connections are required.
- Thus, it is useful to keep all the collections of an application in one database, though a database server can host multiple databases

Query Language

- Unlike the universal English-like SQL in a relational database, the MongoDB query language is made up of methods to achieve various operations.

VTUSYNC.IN

- The main methods for read and write operations are the CRUD methods. Other methods include aggregation, text search, and geospatial queries.
- The query filter is a JavaScript object consisting of zero or more properties, where the property name is the name of the field to match on and the property value consists of another object with an operator and a value.
- For example, to match all documents with the field invoiceNumber that are greater than 1,000, the following query filter can be used:

```
{ "invoiceNumber": { $gt: 1000 } }
```

- Since there is no "language" for querying or updating, the query filters can be very easily constructed programmatically.
- Unlike relational databases, MongoDB encourages denormalization, that is, storing related parts of a document as embedded subdocuments rather than as separate collections (tables) in a relational database.
- Take an example of people (name, gender, etc.) and their contact information (primary address, secondary address etc.).
- In a relational database, this would require separate tables for People and Contacts, and then a join on the two tables when all of the information is needed together.
- In MongoDB, on the other hand, it can be stored as a list of contacts within the same People document. That's because a join of collections is not natural to most methods in MongoDB: the most convenient find() method can operate only on one collection at a time.

VTUSYNC.IN

Installation

Before you try to install MongoDB on your computer, you may want to try out one of the hosted services that give you access to MongoDB. There are many services, but the following are popular and have a free version that you can use for a small test or sandbox application. Any of these will do quite well for the purpose of the Issue Tracker application that we'll build as part of this book.

• MongoDB Atlas (https://www.mongodb.com/cloud/atlas): I refer to this as Atlas for short. A small database (shared RAM, 512 MB storage) is available for free.

• mLab (previously MongoLab) (https://mlab.com/): mLab has announced an acquisition by MongoDB Inc. and may eventually be merged into Atlas itself. A sandbox environment is available for free, limited to 500 MB storage.

• Compose (https://www.compose.com): Among many other services, Compose offers MongoDB as a service. A 30-day trial period is available, but a permanently free sandbox kind of option is not available.

On a Windows system, you may need to append .exe to the command. The command may require a path depending on your installation method. If the shell starts successfully, it will also connect to the local MongoDB server instance. You should see the version of MangoDB printed on the console, the database it is connecting to (the default is test), and a command prompt, like this, if you had installed MongoDB version 4.0.2 locally:

```
MongoDB shell version v4.0.2
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 4.0.2
>
```

VTUSYNC.IN

The Mongo Shell

The mongo shell is an interactive JavaScript shell, very much like the Node.js shell. In the interactive shell, a few non-JavaScript conveniences are available over and above the full power of JavaScript. In this section, we'll discuss the basic operations that are possible via the shell, those that are most commonly used. For a full reference of all the capabilities of the shell, you can take a look at the mongo shell documentation at https://docs.mongodb.com/manual/mongo/.

This will list the databases and the storage occupied by them. For example, in a fresh local installation of MongoDB, this is what you will see:

```
admin           0.000GB
config          0.000GB
local           0.000GB
```

These are system databases that MongoDB uses for its internal book keeping, etc. We will not be using any of these to create our collections, so we'd better change the current database. To identify the current database, the command is:

> db

The default database a mongo shell connects to is called test and that is what you are likely to see as the output to this command. Let's now see what collections exist in this database.

> show collections

You will find that there are no collections in this database, since it is a fresh installation. Further, you will also find that the database test was not listed when we listed the available databases. That's because databases and collections are really created only on the first write operation to any of these. Let's switch to a database called issuetracker instead of using the default database:

> use issuetracker

This should result in output that confirms that the new database is issuetracker:

```
switched to db issuetracker
```

Let's confirm that there are no collections in this database either:

> show collections

This command should return nothing. Now, let's create a new collection. This is done by creating one document in a collection.

VTUSYNC.IN

Apart from the insertOne() method, many methods are available on any collection. You can see the list of available methods by pressing the Tab character twice after typing "db.employees." (the period at the end is required before pressing Tab). You may find an output like the following:

```
db.employees.addIdIfNeeded(                db.employees.getWriteConcern(
db.employees.aggregate(                    db.employees.group(
db.employees.bulkWrite(                    db.employees.groupcmd(
db.employees.constructor                   db.employees.hasOwnProperty
db.employees.convertToCapped(              db.employees.hashAllDocs(
db.employees.convertToSingleObject(        db.employees.help(
db.employees.copyTo(                       db.employees.initializeOrderedBulkOp(
db.employees.count(                        db.employees.initializeUnorderedBulkOp(
db.employees.createIndex(                  db.employees.insert(
db.employees.createIndexes(                db.employees.insertMany(
db.employees.dataSize(                     db.employees.insertOne(
...
```

This is the auto-completion feature of the mongo shell at work. Note that you can let the mongo shell auto-complete the name of any method by pressing the Tab character after entering the beginning few characters of the method.

MongoDB CRUD Operations

CRUD operations in MongoDB refer to the basic database operations:

1. **Create** – Insert new documents into a collection.
   - insertOne(): Inserts a single document.
   - insertMany(): Inserts multiple documents.

   db.employees.insertOne({ id: 1, name: { first: 'John', last: 'Doe' }, age: 44 });
   db.employees.insertMany([{ id: 2, name: { first: 'Jane', last: 'Doe' }, age: 30 }]);

2. **Read** – Retrieve documents from a collection.
   - findOne(): Fetches a single document.
   - find(): Fetches multiple documents with filtering options.

   db.employees.findOne({ id: 1 });

VTUSYNC.IN

```
db.employees.find({ age: { $gte: 30 } });
```

3. **Update** – Modify existing documents.
   - updateOne(): Updates the first matching document.
   - updateMany(): Updates multiple documents.
   - $set: Used to modify specific fields.
   - replaceOne(): Replaces an entire document.

```
db.employees.updateOne({ id: 1 }, { $set: { age: 45 } });
db.employees.updateMany({}, { $set: { organization: 'MyCompany' } });
```

4. **Delete** – Remove documents from a collection.
   - deleteOne(): Deletes a single document.
   - deleteMany(): Deletes multiple documents.

```
db.employees.deleteOne({ id: 4 });
db.employees.deleteMany({ age: { $lt: 20 } });
```

Aggregate

Aggregation in MongoDB is a process of transforming and analyzing data by applying operations such as filtering, grouping, and computations. It allows us to summarize and manipulate data similar to SQL's GROUP BY but with additional capabilities such as joins ($lookup), array expansion ($unwind), and complex transformations.

MongoDB provides the Aggregation Framework, which processes documents through a sequence of pipeline stages. Each stage modifies the data before passing it to the next.

Basic Aggregation Example

The find() method retrieves raw documents, but aggregate() allows for summarization.

1. Summing a Field

To calculate the total age of all employees:

```
db.employees.aggregate([
  { $group: { _id: null, total_age: { $sum: '$age' } } }
```

VTUSYNC.IN

])

Output:

{ "_id" : null, "total_age" : 103 }

2. Counting Documents

To count the number of employees:

```
db.employees.aggregate([
  { $group: { _id: null, count: { $sum: 1 } } }
])
```

Output:

{ "_id" : null, "count" : 3 }

3. Grouping Data by a Field

Let's group employees by their organization and calculate the total age for each:

```
db.employees.aggregate([
  { $group: { _id: '$organization', total_age: { $sum: '$age' } } }
])
```

Output:

{ "_id" : "OtherCompany", "total_age" : 64 }
{ "_id" : "MyCompany", "total_age" : 103 }

4. Calculating Average

To compute the average age of employees per organization:

```
db.employees.aggregate([
  { $group: { _id: '$organization', average_age: { $avg: '$age' } } }
])
```

Output:

{ "_id" : "OtherCompany", "average_age" : 64 }
{ "_id" : "MyCompany", "average_age" : 34.33 }

VTUSYNC.IN

Key Features of Aggregation

1. **Pipeline Processing** – Stages process data step-by-step.
2. **Grouping & Summarization** – $group, $sum, $avg, $min, and $max perform statistical operations.
3. **Filtering & Sorting** – $match filters documents, $sort orders them.
4. **Transforming Data** – $project modifies output fields.
5. **Array Processing** – $unwind expands arrays into separate documents.
6. **Joins** – $lookup enables data fetching from other collections.

MongoDB Node.js Driver

The MongoDB Node.js driver is a library that allows applications built with Node.js to connect to a MongoDB database, interact with collections, and perform CRUD operations. It is similar to using the MongoDB shell but with an API that integrates into JavaScript and Node.js applications.

MongoDB provides a low-level driver as well as an Object Document Mapper (ODM) like Mongoose. The low-level driver offers direct control over the database operations, making it a good choice for understanding how MongoDB works at its core. However, ODMs like Mongoose add a layer of abstraction and provide a more structured approach.

VTUSYNC.IN

1. Installing the MongoDB Node.js Driver

To use the MongoDB driver, first, install it in your Node.js application:

npm install mongodb@3

2. Connecting to a MongoDB Server

To connect to MongoDB, you need to use the MongoClient object. The connection URL specifies the server addr**ess** and the databa**s**e.

Basic Connection Code

```
const { MongoClient } = require('mongodb');
const url = 'mongodb://localhost/issuetracker'; // Local MongoDB URL

const client = new MongoClient(url, { useNewUrlParser: true });

client.connect((err, client) => {
  if (err) {
    console.error("Error connecting to MongoDB:", err);
    return;
  }
  console.log('Connected to MongoDB');

  const db = client.db(); // Get the database instance
  client.close(); // Close the connection after operations
});
```

Connection URL Format

- Local MongoDB:

  mongodb://localhost/issuetracker

- Cloud-based MongoDB Atlas:

  mongodb+srv://UUU:PPP@cluster0-XXX.mongodb.net/issuetracker?retryWrites=true

  - Replace UUU with the username, PPP with the password, and XXX with the cluster hostname.

VTUSYNC.IN

3. Performing CRUD Operations

Once connected, we can interact with collections using the database object (db)**.**

3.1 Inserting a Document

The insertOne() method is used to add a document to a collection. This method is asynchronous and requires a callback function or async/await.

```
const collection = db.collection('employees'); // Get the collection

const employee = { id: 1, name: 'A. Callback', age: 23 };

collection.insertOne(employee, (err, result) => {
   if (err) {
      console.error("Error inserting document:", err);
      return;
   }
   console.log('Inserted Document ID:', result.insertedId);
});
```

3.2 Querying (Finding) Documents

To retrieve documents, use find() followed by .toArray().

```
collection.find({ _id: result.insertedId }).toArray((err, docs) => {
   if (err) {
      console.error("Error retrieving documents:", err);
      return;
   }
   console.log('Found Documents:', docs);
});
```

3.3 Updating a Document

Use updateOne() to modify a document.

```
collection.updateOne({ id: 1 }, { $set: { age: 30 } }, (err, result) => {
   if (err) {
      console.error("Error updating document:", err);
      return;
   }
   console.log('Modified Count:', result.modifiedCount);
});
```

VTUSYNC.IN

3.4 Deleting a Document

To delete a document, use deleteOne().

```
collection.deleteOne({ id: 1 }, (err, result) => {
   if (err) {
      console.error("Error deleting document:", err);
      return;
   }
   console.log('Deleted Count:', result.deletedCount);
});
```

4. Using Async/Await Instead of Callbacks

The callback-based approach can lead to "callback hell" due to deeply nested functions. A cleaner approach is using async/await.

Refactored Code with Async/Await

```
async function testWithAsync() {
   console.log('\n--- testWithAsync ---');
   const client = new MongoClient(url, { useNewUrlParser: true });

   try {
      await client.connect(); // Wait for the connection to establish
      console.log('Connected to MongoDB');

      const db = client.db();
      const collection = db.collection('employees');

      const employee = { id: 2, name: 'B. Async', age: 16 };

      const result = await collection.insertOne(employee);
      console.log('Inserted Document ID:', result.insertedId);

      const docs = await collection.find({ _id: result.insertedId }).toArray();
      console.log('Found Documents:', docs);

   } catch (err) {
      console.error("Error:", err);
```

VTUSYNC.IN

```
    } finally {
        client.close();
    }
}

// Call the function
testWithAsync();
```

Why Use Async/Await?

- Cleaner and more readable code.
- Avoids deeply nested callbacks.
- Uses try...catch for better error handling.

VTUSYNC.IN

5. Running the MongoDB Script

To run the script, save it as trymongo.js and execute:

node scripts/trymongo.js

Before running, clear the collection to prevent duplicate key errors:

mongo issuetracker --eval "db.employees.remove({})"

For MongoDB Atlas:

mongo "mongodb+srv://cluster0-xxxxx.mongodb.net/issuetracker" --username atlasUser --password atlasPassword --eval "db.employees.remove({})"

6. Error Handling

Errors in MongoDB operations should be handled properly using:

1. **Callback-based error handling**

```
if (err) {
    console.error("Error:", err);
    client.close();
    return;
}
```

2. **Using Try-Catch in Async/Await**

```
try {
   await client.connect();
} catch (err) {
   console.error("Connection error:", err);
}
```

7. Indexing and Unique Constraints

MongoDB allows indexing for performance optimization. If a unique index exists on a field, duplicate inserts will fail.

To create a unique index:

db.employees.createIndex({ id: 1 }, { unique: true })

Schema Initialization,

MongoDB is a NoSQL database, meaning it does not enforce a fixed schema like relational databases (SQL-based). However, initializing a schema in MongoDB typically refers to setting up indexes and inserting initial data into collections.

Why Schema Initialization?

Since MongoDB does not require predefined schemas, schema initialization in this context means:

1. **Clearing old data** – Removing all existing documents from a collection.
2. **Inserting default data** – Adding sample data for testing or initial use.
3. **Creating indexes** – Optimizing searches by indexing key fields.

Steps for Schema Initialization

1. Remove Existing Data

Before inserting fresh data, we clear the issues collection to remove any existing documents:

VTUSYNC.IN

```
db.issues.remove({});
```

This ensures that the database starts with a clean state.

2. Insert Initial Data

A predefined array of issue objects is inserted using insertMany(), providing a structured starting point.

```
const issuesDB = [
  {
    id: 1, status: 'New', owner: 'Ravan', effort: 5,
    created: new Date('2019-01-15'), due: undefined,
    title: 'Error in console when clicking Add',
  },
  {
    id: 2, status: 'Assigned', owner: 'Eddie', effort: 14,
    created: new Date('2019-01-16'), due: new Date('2019-02-01'),
    title: 'Missing bottom border on panel',
  },
];
db.issues.insertMany(issuesDB);
```

This helps developers by providing sample data for testing the application.

3. Create Indexes

Indexes are essential for improving query performance. The script creates indexes on:

- id (unique index) to prevent duplicate issue entries.
- status, owner, and created fields for faster filtering and sorting.

```
db.issues.createIndex({ id: 1 }, { unique: true });
db.issues.createIndex({ status: 1 });
db.issues.createIndex({ owner: 1 });
db.issues.createIndex({ created: 1 });
```

VTUSYNC.IN

How to Run the Initialization Script

The script can be executed in different environments:

1. **Local MongoDB instance:**

   mongo issuetracker scripts/init.mongo.js

2. **MongoDB Atlas (Cloud Database):**

   mongo mongodb+srv://user:pwd@xxx.mongodb.net/issuetracker scripts/init.mongo.js

3. **mLab (Hosted MongoDB Service):**

   mongo mongodb://user:pwd@xxx.mlab.com:33533/issuetracker scripts/init.mongo.js

Reading from MangoDB

In a Node.js application, reading data from MongoDB involves establishing a persistent connection to the database and querying collections for relevant data. This process ensures that the application retrieves real-time data from the database instead of relying on an in-memory array.

Steps for Reading from MangoDB

1. Establishing a Persistent Database Connection

Instead of opening and closing the database connection for each query, we maintain a global connection variable (db). This improves efficiency and ensures smooth handling of multiple API requests.

Define the MongoDB Connection URL

```
const { MongoClient } = require('mongodb');

const url = 'mongodb://localhost/issuetracker';  // Local Database
// For Cloud Databases, use:
// const url = 'mongodb+srv://user:password@xxx.mongodb.net/issuetracker';
```

VTUSYNC.IN

Create an Asynchronous Function to Connect

We use the MongoClient from the mongodb package to connect to the database asynchronously:

```
let db;

async function connectToDb() {
  const client = new MongoClient(url, { useNewUrlParser: true });
  await client.connect();  // Connect to MongoDB
  console.log('Connected to MongoDB at', url);
  db = client.db();  // Store database connection in a global variable
}
```

2. Modifying the API to Read from MongoDB

Once the connection is established, we update the API function issueList() to retrieve data from MongoDB instead of an in-memory array.

Updating the issueList() Function

```
async function issueList() {
  const issues = await db.collection('issues').find({}).toArray();
  return issues;
}
```

- .collection('issues') – Accesses the issues collection in MongoDB.
- .find({}) – Retrieves all documents.
- .toArray() – Converts the result to an array.

3. Starting the Server After Database Connection

Since connectToDb() is an asynchronous function, we need to ensure that the database connection is established before the server starts.

Using an Immediately Invoked Async Function

```
(async function () {
  try {
    await connectToDb();  // Connect to MongoDB first
```

VTUSYNC.IN

```
    app.listen(3000, function () {
      console.log('App started on port 3000');
    });
  } catch (err) {
    console.log('ERROR:', err);
  }
})();
```

This ensures:

- The database is connected before the server starts.
- Any connection errors are caught and logged.

4. Updating the GraphQL Schema

Since MongoDB automatically adds an _id field to each document, we update the GraphQL schema to include this field:

```
type Issue {
  _id: ID!
  id: Int!
  ...
}
```

This allows clients to access _id when querying the API.

5. Verifying Data is Read from MongoDB

- Run the server.
- Open the UI and check if the issues are displayed.
- Modify data in MongoDB using the shell:

  ```
  db.issues.updateMany({}, { $set: { effort: 100 } });
  ```

- Refresh the UI – the updated values should reflect.

VTUSYNC.IN

Writing to MangoDB

To fully integrate MongoDB into the application, we need to update the Create API so that it writes new issues directly to the database instead of storing them in an in-memory array.

Steps for Writing to MangoDB

1. Generating Unique IDs for Issues

Unlike relational databases, MongoDB does not provide built-in auto-increment sequences. To ensure unique, sequential IDs for issues, we implement a **c**ounter **c**ollection.

Initialize the Counter in MongoDB

Before using the counter, we initialize it in init.mongo.js:

```
print('Inserted', count, 'issues');
db.counters.remove({ _id: 'issues' });  // Remove any existing counter
db.counters.insert({ _id: 'issues', current: count });  // Initialize counter
```

- The counter document stores the last assigned issue ID.
- When a new issue is added, the counter is incremented, ensuring unique IDs.

Run the Initialization Script

After modifying init.mongo.js, execute:

```
mongo issuetracker scripts/init.mongo.js
```

2. Implementing a Function to Generate Unique IDs

To increment and fetch the latest issue ID, we define getNextSequence() using MongoDB's findOneAndUpdate**()** method.

Function to Get the Next Sequence Number

```
async function getNextSequence(name) {
   const result = await db.collection('counters').findOneAndUpdate(
      { _id: name },   // Find the counter by its name
      { $inc: { current: 1 } },  // Increment the counter by 1
```

VTUSYNC.IN

```
    { returnOriginal: false }  // Return the updated document
  );
  return result.value.current;  // Return the new issue ID
}
```

- **findOneAndUpdate()** ensures atomic updates.
- **$inc: { current: 1 }** increases the counter.
- **returnOriginal: false** makes sure we get the updated value.

3. Updating the Create API to Write to MongoDB

Instead of pushing new issues into an in-memory array, we:

1. **Get a new unique ID using getNextSequence()**.
2. **Insert the new issue into the MongoDB collection**.
3. **Retrieve and return the newly created issue**.

Updating issueAdd() Function

```
async function issueAdd(_, { issue }) {
  issue.created = new Date();  // Set creation timestamp
  issue.id = await getNextSequence('issues');  // Generate unique ID

  const result = await db.collection('issues').insertOne(issue);  // Insert into
MongoDB

  const savedIssue = await db.collection('issues')
    .findOne({ _id: result.insertedId });  // Retrieve the saved issue

  return savedIssue;  // Return the newly added issue
}
```

How This Works

1. **Generate a new ID** → getNextSequence('issues')
2. **Insert issue into MongoDB** → insertOne(issue)
3. **Fetch the inserted document** → findOne({ _id: result.insertedId })
4. **Return the inserted issue** → Ensures the API response includes database-generated fields.

VTUSYNC.IN

4. Removing the In-Memory Array

Since we now store issues in MongoDB, we can remove the in-memory array:

```
const issuesDB = [  // This is no longer needed
    { id: 1, status: 'New', owner: 'Ravan', effort: 5, ... },
    { id: 2, status: 'Assigned', owner: 'Eddie', effort: 14, ... },
];
```

5. Testing the Changes

Check if the API Works

1. **Run the server**

   node server.js

2. **Add a new issue from the UI**
3. **Check if the issue persists after a server restart**

Cross-Check Using MongoDB Shell

To verify that the issue is saved in MongoDB:

```
mongo
use issuetracker
db.issues.find().pretty()
```

This should display all issues, including the newly created one.

VTUSYNC.IN

Modularization and Webpack

We started to get organized in the previous chapter by changing the architecture and adding checks for coding standards and best practices. In this chapter, we'll take this a little further by splitting the code into multiple files and adding tools to ease the process of development. We'll use Webpack to help us split frontend code into component-based files, inject code into the browser incrementally, and refresh the browser automatically on front-end code changes. That's a perfectly valid thought if you are not too concerned about all these, and instead rely on someone else to give you a template of sorts that predefines the directory structure as well as has configuration for the build tools such as Webpack. This can let you focus on the MERN stack alone, without having to deal with all the tooling. In that case, you have the following options:

- Download the code from the book's GitHub repository (https://github.com/vasansr/pro-mern-stack-2) as of the end of this chapter and use that as your starting point for your project.
- Use the starter-kit create-react-app (https://github.com/facebook/createreact-app) to start your new React app and add code for your application. But note that create-react-app deals only with the React part of the MERN stack; you will have to deal with the APIs and MongoDB yourself.
- Use mern.io (http://mern.io) to create the entire application's directory structure, which includes the entire MERN stack.s

VTUSYNC.IN

Back-End Modules

Backend modules help organize and structure code for maintainability, scalability, and reusability. In a Node.js application, modularization is achieved using the require() function for importing and module. exports for exporting functionalities. Below is an explanation of key modules used in the Issue Tracker API, along with how they interact.

1. graphql_date.js (Custom GraphQL Scalar Type)

This module defines a custom GraphQL scalar type for handling date values. It:

- Imports required GraphQL modules.
- Defines a GraphQLScalarType for Date.
- Exports the date scalar for use in the GraphQL schema.

Code:

```
const { GraphQLScalarType } = require('graphql');
const { Kind } = require('graphql/language');

const GraphQLDate = new GraphQLScalarType({
  // Implementation details...
});

module.exports = GraphQLDate;
```

2. about.js (Handling API Information)

This module manages the "about" message, which describes the API version. It:

- Stores the aboutMessage.
- Defines two functions: getMessage() (retrieves the message) and setMessage() (updates it).
- Exports these functions.

Code:

```
let aboutMessage = 'Issue Tracker API v1.0';

function setMessage(_, { message }) {
  aboutMessage = message;
}

function getMessage() {
  return aboutMessage;
}
```

VTUSYNC.IN

```
module.exports = { getMessage, setMessage };
```

3. db.js (Database Connection and ID Generation)

This module handles MongoDB connectivity and ID sequence management. It:

- Connects to MongoDB using connectToDb().
- Provides a function getNextSequence() to generate unique issue IDs.
- Exports these functions along with getDb() to get the database connection.

Code:

```
require('dotenv').config();
const { MongoClient } = require('mongodb');

let db;

async function connectToDb() {
  const url = process.env.DB_URL || 'mongodb://localhost/issuetracker';
  const client = new MongoClient(url, { useNewUrlParser: true });
  await client.connect();
  db = client.db();
}

async function getNextSequence(name) {
  const result = await db.collection('counters').findOneAndUpdate(
    { _id: name },
    { $inc: { current: 1 } },
    { returnOriginal: false }
  );
  return result.value.current;
}

function getDb() {
  return db;
}

module.exports = { connectToDb, getNextSequence, getDb };
```

VTUSYNC.IN

4. issue.js (Issue Management)

This module handles CRUD operations related to issues. It:

- Uses getDb() to get the database connection.
- Provides list() to retrieve all issues.
- Provides add() to create a new issue.

Code:

```
const { UserInputError } = require('apollo-server-express');
const { getDb, getNextSequence } = require('./db.js');

async function list() {
  const db = getDb();
  return await db.collection('issues').find({}).toArray();
}

async function add(_, { issue }) {
  const db = getDb();
  issue.id = await getNextSequence('issues');
  issue.created = new Date();
  const result = await db.collection('issues').insertOne(issue);
  return db.collection('issues').findOne({ _id: result.insertedId });
}

module.exports = { list, add };
```

5. api_handler.js (GraphQL Schema and Apollo Server)

This module sets up the GraphQL schema and resolver functions. It:

- Imports necessary resolvers from graphql_date.js, about.js, and issue.js.
- Creates an Apollo Server instance with the schema.
- Exports the function installHandler() to integrate GraphQL with Express.

Code:

```
const { ApolloServer } = require('apollo-server-express');
const GraphQLDate = require('./graphql_date.js');
```

VTUSYNC.IN

```
const about = require('./about.js');
const issue = require('./issue.js');

const resolvers = {
  Query: {
    about: about.getMessage,
    issueList: issue.list,
  },
  Mutation: {
    setAboutMessage: about.setMessage,
    issueAdd: issue.add,
  },
  GraphQLDate,
};

const server = new ApolloServer({ typeDefs, resolvers });

function installHandler(app) {
  server.applyMiddleware({ app, path: '/graphql' });
}

module.exports = { installHandler };
```

6. **server.js** (Starting the Express Server)

This is the entry point of the application. It:

- Imports required modules.
- Connects to the database.
- Sets up the Express server.
- Calls installHandler() to integrate GraphQL.

Code:

```
require('dotenv').config();
const express = require('express');
const { connectToDb } = require('./db.js');
const { installHandler } = require('./api_handler.js');
```

VTUSYNC.IN

```
const app = express();
installHandler(app);

const port = process.env.API_SERVER_PORT || 3000;

(async function () {
  try {
    await connectToDb();
    app.listen(port, () => console.log(`API server started on port ${port}`));
  } catch (err) {
    console.error('ERROR:', err);
  }
})();
```

VTUSYNC.IN

Front-End Modules and Webpack

In modern web applications, managing large JavaScript files becomes challenging as the project grows. Traditionally, developers used multiple JavaScript files, including them in an HTML file using <script> tags. However, this approach required careful manual dependency management, making it error-prone and difficult to scale.

To solve this, tools like Webpack and Browserify automate dependency resolution, allowing developers to write modular code while bundling everything into optimized JavaScript files.

Why Use Modules in Front-End Development?

1. Maintainability:

- Breaking the code into smaller, reusable modules makes it easier to understand and manage.

2. Dependency Management:

- Webpack automatically determines module dependencies and includes them in the final bundle.

3. Performance Optimization:

- Webpack minifies and optimizes JavaScript, reducing load times.

4. Scalability:

- Modular code allows teams to collaborate efficiently and extend the application easily.

Setting Up Webpack

Since Webpack is only needed for development, we install it as a **dev dependency**:

```
cd ui
npm install --save-dev webpack@4 webpack-cli@3
```

To verify the installation:

```
npx webpack --version
```

Bundling JavaScript Using Webpack

Let's bundle the App.js file into app.bundle.js using Webpack:

```
npx webpack public/App.js --output public/app.bundle.js
```

However, Webpack gives a warning about the missing **mode** option. To remove the warning, specify **development mode**:

```
npx webpack public/App.js --output public/app.bundle.js --mode development
```

In **development mode**, Webpack keeps the code readable for debugging.
In **production mode**, Webpack minifies and optimizes the code.

Splitting Code into Modules

Instead of writing everything in one file, we **split** the code into separate modules.

For example, we extract the graphQLFetch function into a new file.

VTUSYNC.IN

Step 1: Create graphQLFetch.js

graphQLFetch.js

```
const dateRegex = new RegExp('^\\d\\d\\d\\d-\\d\\d-\\d\\d');

function jsonDateReviver(key, value) {
  if (dateRegex.test(value)) return new Date(value);
  return value;
}

export default async function graphQLFetch(query, variables = {}) {
  // Fetch implementation here
}
```

Step 2: Import the Module in App.jsx

Instead of defining the function inside App.jsx, we import it:

```
import graphQLFetch from './graphQLFetch.js';
```

ES6 Modules in Webpack
Importing Modules

- **Using require() (CommonJS - Older Method)**:

  ```
  const graphQLFetch = require('./graphQLFetch.js');
  ```

- **Using import (ES6 - Modern Method)**:

  ```
  import graphQLFetch from './graphQLFetch.js';
  ```

  - Webpack resolves dependencies automatically.

Exporting Modules

- **Named Export** (for multiple functions):

  ```
  export function graphQLFetch() { ... }
  ```

  - Imported as:

VTUSYNC.IN

import { graphQLFetch } from './graphQLFetch.js';

- **Default Export** (for a single function):

  export default function graphQLFetch() { ... }

  - Imported as:

    import graphQLFetch from './graphQLFetch.js';

Updating index.html to Use the Bundle

Since Webpack generates app.bundle.js, update index.html to use it instead of App.js:

```
<script src="/env.js"></script>
<script src="/app.bundle.js"></script>
```

This ensures the browser loads the bundled file containing all required modules.

Final Webpack Build Process
Steps:

1. **Compile JSX using Babel** (if required):

   npm run compile

2. **Run Webpack to bundle files**:

   npx webpack public/App.js --output public/app.bundle.js --mode development

3. **Test the application** in the browser.

Benefits of Using Webpack

Automatic Dependency Management – No need to manually track script order.
Improved Performance – Bundling reduces HTTP requests and optimizes code.

VTUSYNC.IN

Code Splitting – Supports loading parts of the application asynchronously.
Faster Development – Webpack watches for changes and updates the bundle automatically.
Handles Static Assets – Can bundle CSS, images, and fonts.

1. Transform & Bundle with Webpack

- Webpack combines transformation and bundling using **loaders** (e.g., babel-loader for Babel).
- Installed Babel loader:

```
npm install --save-dev babel-loader@8
```

2. **Webpack Configuration (webpack.config.js)**
   - **Entry**: Specifies the main file (App.jsx).
   - **Output**: Generates app.bundle.js in the public directory.
   - **Module Rules**: Uses Babel loader to transform .jsx and .js files.
   - Example:

```
const path = require('path');
module.exports = {
  mode: 'development',
  entry:  './src/App.jsx',
  output: {
    filename: 'app.bundle.js',
    path: path.resolve(__dirname, 'public'),
  },
  module: {
    rules: [
      { test: /\.jsx?$/, use: 'babel-loader' },
    ],
  },
};
```

3. **Running Webpack**
   - Compile once:

```
npx webpack
```

VTUSYNC.IN

- Watch for changes:

  npx webpack --watch

4. **Updating npm Scripts (package.json)**
   - **Production Build**: "compile": "webpack --mode production"
   - **Watch Mode**: "watch": "webpack --watch"
5. **Modularizing Components**
   - **Separated Components**:
     - IssueList.jsx, IssueFilter.jsx, IssueTable.jsx, IssueAdd.jsx, graphQLFetch.js
   - **App.jsx** now only imports IssueList.jsx and renders it.
6. **Final Testing**
   - Running npm run watch ensures all files are transformed and bundled.
   - The application should work as before but with improved modularity.

Bundling Libraries with Webpack

Previously, we included third-party libraries (like React, ReactDOM, and Babel polyfills) using CDNs in **index.html**. However, this approach has drawbacks:

- **Dependency on CDN availability** – If the CDN is down, the app won't load.
- **Performance issues** – Every time the app loads, the browser fetches these libraries.
- **No caching benefit** – Even if only the app code changes, all scripts are refetched.

Installing Client-Side Libraries with npm

To manage these libraries within our project, we install them using npm:

```
$ cd ui
$ npm install react@16 react-dom@16
$ npm install prop-types@15
$ npm install whatwg-fetch@3
$ npm install babel-polyfill@6
```

After installation, we **import** these libraries in our React files instead of relying on CDNs.

VTUSYNC.IN

Optimizing Bundling in Webpack

1. **Single Bundle Issue**
   Initially, Webpack bundled everything (including libraries) into app.bundle.js, making the file large (>1MB). This meant that even minor application changes required users to re-download the entire file.
2. **Creating Separate Bundles**
   To improve efficiency, we split the libraries into a separate **vendor bundle** using Webpack's splitChunks optimization. This results in:
   - vendor.bundle.js (for third-party libraries)
   - app.bundle.js (for our application code)
3. **Webpack Configuration Changes**
   - **Exclude node_modules** from transformations since they are already optimized.
   - **Modify entry and output** to support multiple bundles.
   - **Enable splitChunks** to separate dependencies.

```
module.exports = {
  mode: 'development',
  entry: { app: './src/App.jsx' },
  output: {
    filename: '[name].bundle.js', // Generates app.bundle.js and
vendor.bundle.js
    path: path.resolve(__dirname, 'public'),
  },
  module: {
    rules: [
      {
        test: /\.jsx?$/,
        exclude: /node_modules/,
        use: 'babel-loader',
      },
    ],
  },
  optimization: {
    splitChunks: {
    name: 'vendor',
    chunks: 'all',
    },
  },
```

VTUSYNC.IN

```
};
```

4. **Updating index.html**
   - **Remove CDN links**.
   - **Include vendor.bundle.js before app.bundle.js**.

```
<script src="/vendor.bundle.js"></script>
<script src="/app.bundle.js"></script>
```

Benefits of Splitting Bundles

- **Performance Boost**: The browser caches vendor.bundle.js, so changes to application code only require reloading app.bundle.js.
- **Reduced Load Times**: Instead of re-downloading everything, only the necessary updates are fetched.
- **Offline Resilience**: The app is not dependent on CDNs for libraries.

Hot Module Replacement

Problem with Webpack Watch Mode

- Webpack's watch mode detects file changes and recompiles, but **you must manually refresh** the browser to see updates.
- Refreshing **too soon** might load an outdated bundle.
- Requires **an extra terminal** for running npm run watch.

Solution: Hot Module Replacement (HMR)

HMR updates **only the changed modules** in the browser **without refreshing the entire page**, preserving application state.

Key Benefits:

- **No need to refresh manually**
- **Retains state** (e.g., text in input fields stays)
- **Saves time** by only updating changed code

VTUSYNC.IN

⬭ Implementing HMR in an Express-based UI Server

1. **Install Middleware Packages**

   npm install --save-dev webpack-dev-middleware@3 webpack-hot-middleware@2

2. **Modify webpack.config.js**
   - Change entry point to an **array** to add HMR support:

     entry: { app: ['./src/App.jsx'] }

   - Add HMR entry point:

     config.entry.app.push('webpack-hot-middleware/client');

   - Enable HMR plugin:

     config.plugins.push(new webpack.HotModuleReplacementPlugin());

3. **Modify uiserver.js to Include HMR Middleware**

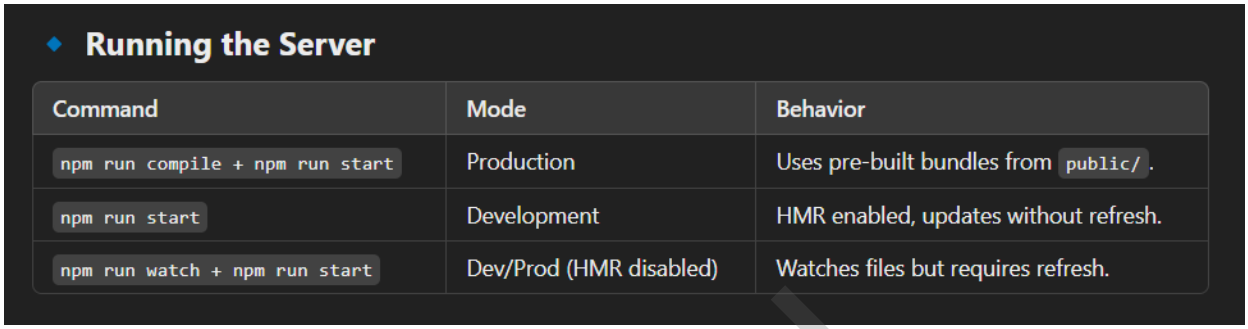
```
if (enableHMR && process.env.NODE_ENV !== 'production') {
  const webpack = require('webpack');
  const devMiddleware = require('webpack-dev-middleware');
  const hotMiddleware = require('webpack-hot-middleware');
  const config = require('./webpack.config.js');

  config.entry.app.push('webpack-hot-middleware/client');
  config.plugins = config.plugins || [];
  config.plugins.push(new webpack.HotModuleReplacementPlugin());

  const compiler = webpack(config);
  app.use(devMiddleware(compiler));
  app.use(hotMiddleware(compiler));
}
```

4. **Modify App.jsx to Accept HMR**

```
if (module.hot) {
  module.hot.accept();
```

VTUSYNC.IN

}



🔹 **Running the Server**

Command	Mode	Behavior
`npm run compile + npm run start`	Production	Uses pre-built bundles from `public/`.
`npm run start`	Development	HMR enabled, updates without refresh.
`npm run watch + npm run start`	Dev/Prod (HMR disabled)	Watches files but requires refresh.

Handling Component Reload Issues

- By default, HMR reloads the **entire React component tree**, **losing local state**.
- **Solution:** Use **react-hot-loader** (not implemented in this case).

Debugging DefinePlugin:

- The unpleasant thing about compiling files is that the original source code gets lost, and if you have to put breakpoints in the debugger, it's close to impossible, because the new code is hardly like the original.
- Creating a bundle of all the source files makes it worse, because you don't even know where to start. Fortunately, Webpack solves this problem by its ability to give you source maps, things that contain your original source code as you typed it in.
- The source maps also connect the line numbers in the transformed code to your original code. Browsers' development tools understand source maps and correlate the two, so that breakpoints in the original source code are converted breakpoints in the transformed code.
- Webpack configuration can specify what kind of source maps can be created along with the compiled bundles. A single configuration parameter called devtool does the job.
- The kind of source maps that can be produced varies, but the most accurate (and the slowest) is generated by the value source-map. For this application,

VTUSYNC.IN

because the UI code is small enough, this is not discernably slow, so let's use it as the value for devtool. The changes to webpack.config.js in the UI directory are shown in Listing 8-29

Listing 8-29. ui/webpack.config.js: Enable Source Map

```
...
  optimization: {
    ...
  },
  devtool: 'source-map'
};
...
```

If you are using the HMR-enabled UI server, you should see the following output in the console that is running the UI server:

```
webpack built dc6a1e03ee249e546ffb in 2964ms
[wdm]: Hash: dc6a1e03ee249e546ffb
Version: webpack 4.23.1
Time: 2964ms
Built at: 10/27/2018 12:08:12 AM
               Asset      Size  Chunks             Chunk Names
       app.bundle.js  54.2 KiB     app  [emitted]  app
   app.bundle.js.map  41.9 KiB     app  [emitted]  app
    vendor.bundle.js  1.26 MiB  vendor  [emitted]  vendor
vendor.bundle.js.map   1.3 MiB  vendor  [emitted]  vendor
Entrypoint app = vendor.bundle.js vendor.bundle.js.map app.bundle.js app.bundle.js.map
[0] multi ./src/App.jsx webpack-hot-middleware/client 40 bytes {app} [built]
[./node_modules/ansi-html/index.js] 4.16 KiB {vendor} [built]
[./node_modules/babel-polyfill/lib/index.js] 833 bytes {vendor} [built]
...
```

VTUSYNC.IN

As you can see, apart from the package bundles, there are accompanying maps with the extension .map. Now, when you look at the browser's Developer Console, you will be able to see the original source code and be able to place breakpoints in it. A sample of this in the Chrome browser is shown in Figure 8-1

Figure 8-1. *Breakpoints in the original source code using source maps*

If you are using the Chrome or Firefox browser, you will also see a message in the console asking you to install the React Development Tools add-on. You can find installation instructions for these browsers at https://reactjs.org/blog/2015/09/02/new-react-developer-tools.html. This addon provides the ability to see the React components in a hierarchical manner like the DOM inspector. For example, in the Chrome browser, you'll find a React tab in the developer tools. Figure 8-2 shows a screenshot of this add-on.

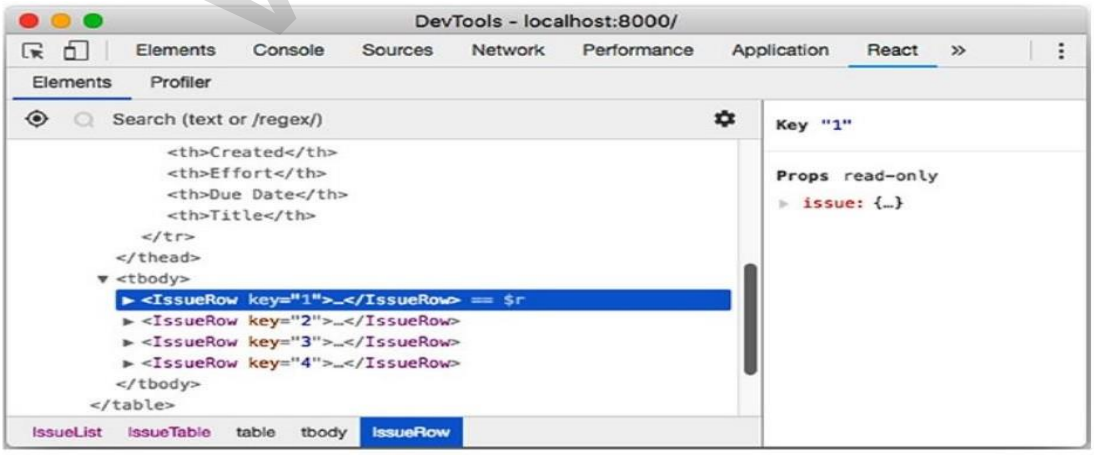


Figure 8-2. *React Developer Tools in the Chrome browser*

VTUSYNC.IN

Define Plugin: Build Configuration

You may not be comfortable with the mechanism that we used for injecting the environment variables in the front-end: a generated script like env.js. For one, this is less efficient than generating a bundle that already has this variable replaced wherever it needs to be replaced. The other is that a global variable is normally frowned upon, since it can clash with global variables from other scripts or packages. Fortunately, there is an option. We will not be using this mechanism for injecting environment variables, but I have discussed it here so that it gives you an option to try out and adopt if convenient.

To replace variables at build time, Webpack's DefinePlugin plugin comes in handy. As part of webpack.config.js, the following can be added to define a predefined string with the value like this:

```
...
  plugins: [
    new webpack.DefinePlugin({
      __UI_API_ENDPOINT__: "'http://localhost:3000/graphql'",
    })
  ],
...
```

Now, within the code for `App.jsx`, instead of hard-coding this value, the `__UI_API_ENDPOINT__` string can be used like this (note the absence of quotes; it is provided by the variable itself):

```
...
    const response = await fetch(__UI_API_ENDPOINT__, {
...
```

When Webpack transforms and creates a bundle, the variable will be replaced in the source code, resulting in the following:

```
...
    const response = await fetch('http://localhost:3000/graphql', {
...
```

Within `webpack.config.js`, you can determine the value of the variable by using `dotenv` and an environment variable instead of hard-coding it there:

```
...
require('dotenv').config();
...
    new webpack.DefinePlugin({
      __UI_API_ENDPOINT__: `'${process.env.UI_API_ENDPOINT}'`,
    })
...
```

VTUSYNC.IN

Production Optimization

1 **Bundle Size & Performance**

- Webpack minifies JavaScript in production mode.
- Initial vendor bundle size is small (~200KB) but grows as features are added.
- Large bundle sizes can trigger Webpack warnings about performance impact.

2 **Handling Large Bundles**

- **Frequent-use apps (e.g., Issue Tracker):** Browser caching reduces concerns.
- **Infrequent-use apps:** Need better optimization to improve page load times.
- **Solution:** Use **code splitting** and **lazy loading** to load scripts only when required.

3 **Lazy Loading Strategy**

- Load only essential scripts upfront.
- Postpone loading non-critical components until needed.
- Useful in **server-rendered** React apps.

4 **Browser Caching Issues**

- Older browsers (e.g., Internet Explorer) may aggressively cache outdated scripts.
- Modern browsers usually check for updates, but explicit cache-busting may be needed.

5 C**ache Busting Solution**

- Use **content hashes** in script file names to force browsers to load new versions.
- Webpack can generate unique file names based on content changes.

6 **Managing Auto-Generated Script Names**

- Need to update index.html with new script names dynamically.

VTUSYNC.IN

- **Solution:** Use **HTMLWebpackPlugin** to generate index.html automatically.

7 Further Reading

- Webpack Code Splitting: webpack.js.org/guides/code-splitting
- Webpack Lazy Loading: webpack.js.org/guides/lazy-loading
- Webpack Caching: webpack.js.org/guides/caching
- HTMLWebpackPlugin: webpack.js.org/plugins/html-webpack-plugin

VTUSYNC.IN