That's an **excellent decision** and a highly recommended best practice, especially for applications with a significant number of CRUD operations and distinct data domains.

Using modular schemas (type definitions) and modular resolvers will make your GraphQL API much more:

- **Maintainable:** Easier to find and update specific parts of your API without navigating massive files.
- **Scalable:** Simpler to add new features or domains without conflicts.
- **Readable:** Code becomes more organized and easier to understand.
- **Testable:** Smaller, focused modules are generally easier to test.
- **Team-Friendly:** Different team members can work on different modules with less chance of stepping on each other's toes.

Here's how you can typically structure and implement modular schemas and resolvers:

I. Modular Type Definitions (Schemas)

You'll break down your schema into smaller files, usually based on the data entity or domain.

1. **Directory Structure:**
2. src/
3. ├── graphql/
4. │ ├── schemas/ # Directory for modular schema parts
5. │ │ ├── user.schema.js
6. │ │ ├── post.schema.js
7. │ │ ├── comment.schema.js
8. │ │ └── index.js # To combine all schema parts
9. │ ├── resolvers/ # (Your modular resolvers directory)
10. │ └── ...
11. └── ...
12. Individual Schema Files (e.g., user.schema.js):

Each file will define the types, queries, mutations, and potentially inputs related to that specific entity.

JavaScript

```
// src/graphql/schemas/user.schema.js
const { gql } = require('apollo-server-express');

module.exports = gql`
 type User {
  id: ID!
  username: String!
  email: String
  # Add other user-specific fields
  posts: [Post!] # Example of a relationship
 }

 # Inputs for mutations often mirror parts of the type
 input CreateUserInput {
  username: String!
  email: String!
  # other required fields for creation
 }

 input UpdateUserInput {
  username: String
  email: String
  # other fields for update
 }

 # It's good practice to extend Query and Mutation types
 # This allows merging without explicitly naming them in each file
 extend type Query {
```

```
  user(id: ID!): User
  users: [User!]
 }

 extend type Mutation {
  createUser(input: CreateUserInput!): User
  updateUser(id: ID!, input: UpdateUserInput!): User
  deleteUser(id: ID!): Boolean # Or return the deleted User
 }
`;
```

- **extend type Query and extend type Mutation:** This is crucial. By using extend, you tell GraphQL that you're adding fields to the existing root Query and Mutation types. If you just wrote type Query { ... } in each file, you'd be trying to redefine Query multiple times, leading to errors. You'll need a base definition for Query and Mutation somewhere.

13. Combining Schemas (schemas/index.js):

You'll need a root schema definition that might just contain the base Query and Mutation types (and potentially any shared scalar types), and then you'll import and combine all the modular parts.

JavaScript

```
// src/graphql/schemas/index.js
const { gql } = require('apollo-server-express');
const userSchema = require('./user.schema');
const postSchema = require('./post.schema');
const commentSchema = require('./comment.schema');
// ... import other schemas

// Base schema definition (important for 'extend' to work)
// This can also be a good place for any global types or interfaces.
const linkSchema = gql`
```

```
 type Query {
  _empty: String # Placeholder, Apollo Server needs at least one field in Query.
            # This will be extended by your modular schemas.
 }

 type Mutation {
  _empty: String # Placeholder, will be extended.
 }

 # You might also define common scalar types or interfaces here
 # scalar DateTime
`;

// Array of all your schema parts
const typeDefs = [
  linkSchema,
  userSchema,
  postSchema,
  commentSchema,
  // ... add other imported schemas here
];

module.exports = typeDefs; // This will be an array of gql tagged template literals or strings
```

Apollo Server (and graphql-tools) can accept an array of type definition strings/ASTs and will merge them.

II. Modular Resolvers

Similar to schemas, you'll break down resolvers by domain.

1. **Directory Structure:**

2. src/
3. ├── graphql/
4. │ ├── schemas/
5. │ │ └── ...
6. │ ├── resolvers/ # Directory for modular resolvers
7. │ │ ├── user.resolvers.js
8. │ │ ├── post.resolvers.js
9. │ │ ├── comment.resolvers.js
10. │ │ └── index.js # To combine all resolver objects
11. │ └── ...
12. └── ...
13. Individual Resolver Files (e.g., user.resolvers.js):

Each file will contain the resolver functions for the queries, mutations, and type fields defined in its corresponding schema module.

JavaScript

```
// src/graphql/resolvers/user.resolvers.js

const { AppDataSource } = require('../../config/data-source'); // Adjust path as needed

const UserEntity = require('../../entities/User'); // Adjust path


module.exports = {
  Query: {
    user: async (_, { id }) => {
      const userRepository = AppDataSource.getRepository(UserEntity.options.name);
      return await userRepository.findOneBy({ id: parseInt(id) });
    },
    users: async () => {
      const userRepository = AppDataSource.getRepository(UserEntity.options.name);
      return await userRepository.find();
```

```
  },
 },
 Mutation: {
  createUser: async (_, { input }) => {
   const userRepository = AppDataSource.getRepository(UserEntity.options.name);
   const newUser = userRepository.create(input); // input will be { username, email }
   await userRepository.save(newUser);
   return newUser;
  },
  updateUser: async (_, { id, input }) => {
   const userRepository = AppDataSource.getRepository(UserEntity.options.name);
   // Fetch user, update, save. Handle not found cases.
   let user = await userRepository.findOneBy({ id: parseInt(id) });
   if (!user) throw new Error('User not found');
   Object.assign(user, input); // Apply partial updates
   await userRepository.save(user);
   return user;
  },
  deleteUser: async (_, { id }) => {
   const userRepository = AppDataSource.getRepository(UserEntity.options.name);
   const result = await userRepository.delete(parseInt(id));
   return result.affected > 0; // Returns true if a row was deleted
  },
 },
 // If User type has fields that need custom resolution (e.g., computed fields or complex relations)
 User: {
  // Example: if 'posts' on User type wasn't directly on the entity or needed special fetching
```

```
    // posts: async (parentUser) => {
    //   // parentUser is the resolved User object from a higher-level query
    //   const postRepository = AppDataSource.getRepository(PostEntity.options.name);
    //   return await postRepository.findBy({ authorId: parentUser.id });
    // },
  },
};
```

14. Combining Resolvers (resolvers/index.js):

You'll import all your modular resolver objects and merge them. lodash.merge is often used for a deep merge, which is helpful.

JavaScript

```
// src/graphql/resolvers/index.js
const { merge } = require('lodash'); // A popular utility for deep merging
const userResolvers = require('./user.resolvers');
const postResolvers = require('./post.resolvers');
const commentResolvers = require('./comment.resolvers');
// ... import other resolvers

// Merge all resolver objects. Lodash's merge does a deep merge.
const resolvers = merge(
  {}, // Start with an empty object for a clean merge
  userResolvers,
  postResolvers,
  commentResolvers,
  // ... add other imported resolvers here
);

module.exports = resolvers;
```

If you don't want to use lodash.merge, you can also spread them manually, but be careful with deeply nested structures like Query and Mutation:

JavaScript

```
// Manual merge example (simpler, but less robust for deep structures than lodash.merge)
// const resolvers = {
//   Query: {
//     ...userResolvers.Query,
//     ...postResolvers.Query,
//   },
//   Mutation: {
//     ...userResolvers.Mutation,
//     ...postResolvers.Mutation,
//   },
//   User: { ...userResolvers.User }, // if you have type-specific resolvers
//   Post: { ...postResolvers.Post },
// };
```

lodash.merge is generally safer as it handles merging nested objects correctly.

III. Updating server.js

Your main server setup file will then import the combined typeDefs and resolvers.

JavaScript

```
// src/server.js
// ... other imports
const { ApolloServer } = require('apollo-server-express');
const typeDefs = require('./graphql/schemas'); // Points to schemas/index.js
const resolvers = require('./graphql/resolvers'); // Points to resolvers/index.js
// ...

async function startServer() {
```

```
  // ... DB Init ...

  const server = new ApolloServer({
    typeDefs,  // This is now an array of schema definitions
    resolvers, // This is your merged resolver object
    // context, etc.
  });

  // ... rest of server setup
}

startServer();
```

This modular approach is very powerful and is the standard way to build larger GraphQL APIs. It keeps your codebase clean, organized, and ready for growth!