ComposerJS

AcceleratXR, Inc.

Aug 28, 2021

CONTENTS

CHAPTER

ONE

ABOUT COMPOSER

The roots of the Composer.js project date back to 2018 when Jean-Philippe Steinmetz began work on a new MMO gaming platform called AcceleratXR. It was decided early on that this project would have to take a micro-service approach in order to reach levels of scalability and throughput required. Each system or feature supported by the platform needed to be built quickly, reusing as much code as possible, while providing a high level of consistency and reliability. It was then that Jean-Philippe decided a code generator tool was the best way to go.

Upon first discovering OpenAPI Jean-Philippe was very excited, especially at the prospect of pre-made code generation tools provided by the community. However, these tools quickly proved inadequate to build the complex and mature platform he was making. So he began to tinker on his own. The first version of Composer (then called `axr-generator`) was crude. The tool would copy a complete working service project from a basic template including all service classes and utilities. This meant that every micro-service project had the same duplicated code. While this was great for productivity it quickly became an enormous burden for code maintainability. If a bug was found in the core code, each project had to be re-generated or merged. This was not a sustainable path for development.

Upon making a transition to TypeScript the tools had begun to evolve and take shape. All common functionality were now placed in libraries that could be added as dependencies and updated with ease. All of the core functionality was abstracted away from the service itself and leveraged runtime file scanning and decorators to parse the desired behavior. The code generator tool was modified to only create the minimum files necessary for a given project; the models and route handlers.

As more and more features were added to the framework it became apparent that the project was becomming unique and special. So Jean-Philippe and the rest of the AcceleratXR team decided to rename the system to Composer.js and release it to the community so that everyone can build professional RESTful API services faster, with more consistency and with enterprise features.

CHAPTER

TWO

FEATURES

Composer comes with a powerful feature set out of the box and is easily extended to provide additional capability.

- Built on OpenAPI
- HTTP 1.x/2.x Web Server
- WebSocket Support
- Global Configuration
- Dependency Injection
- Built-in behaviors for common REST API actions
- Built-in ORM layer
- Document Version Tracking
- MongoDB support
- Redis support
- 2nd Level Caching
- Authentication
- Role Based Access Control
- Prometheseus metrics

CHAPTER

THREE

INSTALLATION

Installing Composer is very easy. From any terminal or command line simply run the following command.

```
yarn global add @composer-js/cli
```

Once complete can now run `composer`.

```
composer --help
```

CHAPTER

FOUR

OPENAPI EXTENSIONS

Every Composer project starts with an OpenAPI specification file. OpenAPI is a standard, language-agnostic, document format for describing RESTful APIs. It is from this document that both server and client code can be easily generated using the CLI tool. This enables developers to save time on initial project creation as well as providing consistent and well-documented APIs for others to use long after the project is deployed.

The OpenAPI specification is somewhat limited, however, and avoids describing certain details about the implementation of a RESTful API. Details that a code generator and framework like Composer must have in order to do its job properly. Therefore, we have extended the specification to add several new features.

Click here for an example.

4.1 Datastore Objects

Holds a set of datastore connection configurations. All objects defined within the `x-datastores` object are copied to the service's `config.ts` file. Schema objects desiring to be bound to a given Datastore Object must explicitly reference it using the `x-datastore` field.

Example

```
components:
    x-datastores:
            mongodb:
                type: mongodb
                url: mongodb://localhost
```

4.2 Datastore Object

Holds the name and associated configuration for a particular datastore connection. All properties defined in the object are copied to the service's `config.ts` file. Configuration options should conform to the TypeORM Connection Options. Desired Schema objects that are to be bound to a given Datastore object must be explicitly referenced with the `x-datastore` field where the value matches the name of the object.

Example

```
components:
    x-datastores:
            mongodb:
                type: mongodb
                url: mongodb://localhost
```

4.3 Schema Object

The following extensions apply to Schema definitions.

4.3.1 x-baseClass

Default Value: `null`

The `x-baseClass` field is applied to a Schema to identify the base class behavior that the generated Schema class will inherit. There are three possible values presently supported by Composer.

Possible Values:

- `BaseMongoEntity` - Provides base behavior for all Schema classes that will be stored in a collection of a MongoDB database.
- `BaseSQLEntity` - Provides base behavior for all Schema classes that will be stored in a SQL database.

Example

```yaml
components:
    schemas:
        Order:
            type: "object"
            x-baseClass: BaseMongoEntity
            x-datastore: mongodb
```

4.3.2 x-datastore

Default Value: `null`

The `x-datastore` field is applied to a Schema to identify which database connection the schema will be bound to. The name of the database connection must match a defined Datastore Object.

Example

```yaml
components:
    schemas:
        Order:
            type: "object"
            x-baseClass: BaseMongoEntity
            x-datastore: mongodb
```

4.3.3 x-ignore

Default Value: `false`

The `x-ignore` field is applied to a Schema Property to indicate that it should be ignored from code generation.

Example

```yaml
components:
    schemas:
        Order:
```

(continues on next page)

(continued from previous page)

```
            type: "object"
            x-ignore: true
```

4.4 Schema Properties

The following extensions have been added to Schema Object Property definitions.

4.4.1 x-identifier

Default Value: `false`

The `x-identifier` field is applied to a Schema Property to indicate that the field is a unique identifier within the database and should be indexed and must be unique.

Example

```
components:
    schemas:
        Order:
            type: "object"
            x-baseClass: BaseMongoEntity
            x-datastore: mongodb
            properties:
                name:
                    type: "string"
                    x-identifier: true
                    x-unique: true
                    nullable: false
```

4.4.2 x-unique

Default Value: `false`

The `x-unique` field is applied to a Schema Property to indicate that the field is a unique identifier within the database and must be unique.

Example

```
components:
    schemas:
        Order:
            type: "object"
            x-baseClass: BaseMongoEntity
            x-datastore: mongodb
            properties:
                name:
                    type: "string"
                    x-identifier: true
                    x-unique: true
                    nullable: false
```

4.5 Path Item Object

The following extensions apply to Path Item Object definitions.

4.5.1 x-name

Default Value: `null`

The `x-name` field is applied to a Path Item Object. It defines the unique name of the path item and the name of the generated code route handler class. This field is superseded by the `x-schema` field.

Example

```yaml
paths:
    /user/login:
        x-name: Auth
        get:
            description: Authenticates the user using HTTP Basic and returns a JSON Web Token access token to be used with future API requests.
            x-name: login
            responses:
                "200":
                    description: The JSON Web Token to be used for all future requests.
                    content:
                        application/json:
                            schema:
                                $ref: "#/components/schemas/authToken"
```

Generated Code

```
/**
 * Handles all REST API requests for the endpoint `/user/login`.
 *
 * @author <AUTHOR>
 */
@Route("/user/login")
class AuthRoute {
    @Config
    protected config: any;
    @Logger
    protected logger: any;

    /**
     * Initializes a new instance with the specified defaults.
     */
    constructor() {
    }
    ...
}
```

4.5.2 x-schema

Default Value: `null`

The `x-schema` field is applied to a Path Item Object. It defines the Schema that the path item is bound to. This supersedes the `x-name` field.

Example

```yaml
paths:
    /pet:
        x-schema: Pet
        get:
            description: "Multiple Pet objects"
            x-name: "find"
            responses:
                "200":
                    description: A list of Pet objects.
                    content:
                        application/json:
                            schema:
                                type: "array"
                                items:
                                    $ref: "#/components/schemas/Pet"
```

Generated Code

```
/**
 * Handles all REST API requests for the endpoint `/pet`.
 *
 * @author <AUTHOR>
 */
@Model(Pet)
@Route("/pet")
class PetRoute extends ModelRoute<Pet> {
    @Config
    protected config: any;
    @Logger
    protected logger: any;

    @MongoRepository(Pet)
    protected repo?: Repo<Pet>;

    /**
     * Initializes a new instance with the specified defaults.
     */
    constructor() {
        super();
    }
    ...
}
```

4.6 Operation Object

The following extensions apply to Operation Object definitions.

4.6.1 x-after

Default Value: `[]`

The `x-after` field is applied to a Operation Object when it is desirable to execute one or more middleware functions **after** the primary endpoint handler has finished. The value is an array of strings, each being the name of the function to execute.

Example

```yaml
/user:
    x-schema: User
    post:
        description: Create a new User.
        x-name: create
        x-before:
        - validate
        x-after:
        - prepareOutput
        requestBody:
            content:
                application/json:
                    schema:
                        $ref: "#/components/schemas/User"
        responses:
            "201":
                description: The newly created User.
                content:
                    application/json:
                        schema:
                            $ref: "#/components/schemas/User"
```

Generated Code

```typescript
/**
 * Create a new User.
 */
@Auth(["jwt"])
@Before(["validate"])
@After(["prepareOutput"])
@Post()
private async create(obj: User, @AuthUser user?: JWTUser): Promise<User> {
    const newObj: User = new User(obj);

    throw new Error("This route is not implemented.");
}
```

4.6.2 x-before

Default Value: `[]`

The `x-before` field is applied to a Operation Object when it is desirable to execute one or more middleware functions **before** the primary endpoint handler has finished. The value is an array of strings, each being the name of the function to execute.

Example

```yaml
/user:
    x-schema: User
    post:
        description: Create a new User.
        x-name: create
        x-before:
        - validate
        x-after:
        - prepareOutput
        requestBody:
            content:
                application/json:
                    schema:
                        $ref: "#/components/schemas/User"
        responses:
            "201":
                description: The newly created User.
                content:
                    application/json:
                        schema:
                            $ref: "#/components/schemas/User"
```

Generated Code

```typescript
/**
 * Create a new User.
 */
@Auth(["jwt"])
@Before(["validate"])
@After(["prepareOutput"])
@Post()
private async create(obj: User, @AuthUser user?: JWTUser): Promise<User> {
    const newObj: User = new User(obj);

    throw new Error("This route is not implemented.");
}
```

4.6.3 x-name

Default Value: `null`

The `x-name` field is applied to a Operation Object. It defines the unique name of the operation and the function name of the generated code for the endpoint handler.

Example

```yaml
paths:
    /pet:
        x-schema: Pet
        get:
            description: "Multiple Pet objects"
            x-name: "find"
            responses:
                "200":
                    description: A list of Pet objects.
                    content:
                        application/json:
                            schema:
                                type: "array"
                                items:
                                    $ref: "#/components/schemas/Pet"
```

Generated Code

```typescript
/**
 * Multiple Pet objects
 */
@Get()
private async find(@AuthUser user?: JWTUser): Promise<Array<Pet>> {
    throw new Error("This route is not implemented.");
}
```

CHAPTER

FIVE

AUTHENTICATION

All Composer projects leverage JSON Web Tokens (JWT) as the primary mechanism for user authentication due to its speed, flexibility and security robustness. As Composer is first and foremost a framework that was designed for building highly scalable, distributed, micro-services and platforms the distributed nature of JWT makes it an easy choice as it eliminates any cross-talk to a central authentication server to validate incoming requests from a user.

5.1 JWT Payload

In order for this to work correctly, all JWT tokens must be formatted in a particular way in order for Composer services to be able to decode them properly. As such the token's payload must contain a `profile` property which contains at least a `uid` to uniquely identify the user as well as an optional list of `roles` which will determine the available permissions the user has to perform actions within the service.

```
{
    "profile": {
        "uid": "3b61f36b-8254-4a7a-ae36-33459692127d",
        "roles": []
    }
}
```

Different services may choose to require additional properties within the `profile` but these are the ones Composer uses internally to perform all of its work.

Optionally, a token's `profile` payload may be encrypted to provide an additional level of security.

5.2 Server Authentication

By default, all Composer services come with a built-in authentication handler for JWT authentication tokens. Tokens may be passed to the service by one of three methods; the Authorization header, the cookie header, or the `jwt_token` query parameter.

5.2.1 Authorization Header

A Composer service will automatically decode a JWT token and attempt to validate it when an `Authorization` header is included with a request containing either the `Bearer` or `jwt` type indicator.

Example

```
GET /path HTTP/1.1
Host: composerjs.io
Authorization: Bearer 0938r2098n209rq38v90bb87209830928r0q3809r8302vqr803q28r0982q0
```

```
GET /path HTTP/1.1
Host: composerjs.io
Authorization: jwt 0938r2098n209rq38v90bb87209830928r0q3809r8302vqr803q28r0982q0
```

5.2.2 Cookie Header

Composer also recognizes cookies with the name `jwt` that are passed in to the HTTP request.

Example

```
GET /path HTTP/1.1
Host: composerjs.io
Cookie: jwt=0938r2098n209rq38v90bb87209830928r0q3809r8302vqr803q28r0982q0
```

5.2.3 Query Parameter

Sometimes it is not possible to pass the authentication token via a header. This is common in debugging scenarios or on platforms that don't allow custom headers for HTTP clients. In such a case you can also pass in the token using the special `jwt_token` query parameter.

Example

```
GET /path?jwt_token=0938r2098n209rq38v90bb87209830928r0q3809r8302vqr803q28r0982q0 HTTP/1.
↪1
Host: composerjs.io
```

CHAPTER

SIX

CREATING A SERVER

The first thing to do is a create the OpenAPI specification file needed to describe the Composer server.

Click here for an example.

Once your spec file is complete, simply run it through the Composer CLI tool, specifying `nodejs` as the language and `server` as the type.

```
composer -i ./service.yaml -o . -t server -l nodejs
```

In the above example we defined our schema in the file `service.yaml` and told Composer to generate a new project in the current directory.

6.1 Building the Server

Before you can build the server project you will need to install the project dependencies.

```
yarn install
```

After that simply run `yarn build`.

```
yarn build
```

6.2 Running the Server

All Composer projects come pre-configured to be run with Docker Compose.

```
docker-compose build
docker-compose run
```

Once the system is up and running you'll see log output such as the following.

```
server_1   | 2020-05-30T23:35:10.725Z info: Registered Route: GET /
server_1   | 2020-05-30T23:35:10.726Z info: Registered Route: POST /acls
server_1   | 2020-05-30T23:35:10.726Z info: Registered Route: DELETE /acls/:id
server_1   | 2020-05-30T23:35:10.726Z info: Registered Route: GET /acls
server_1   | 2020-05-30T23:35:10.726Z info: Registered Route: GET /acls/:id
server_1   | 2020-05-30T23:35:10.726Z info: Registered Route: PUT /acls/:id
server_1   | 2020-05-30T23:35:10.729Z info: Registered Route: GET /metrics
```

(continues on next page)

(continued from previous page)

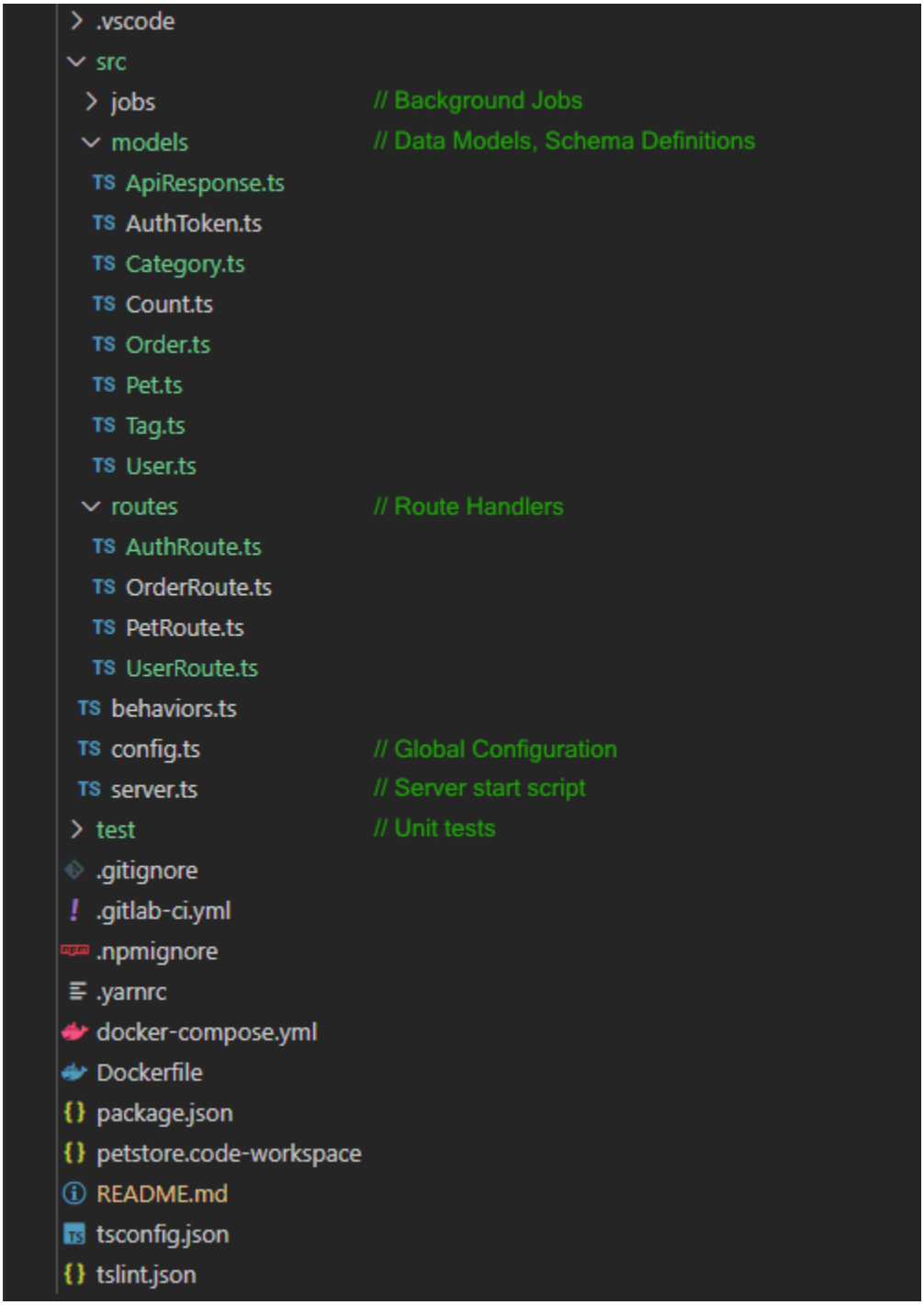
```
server_1   | 2020-05-30T23:35:10.729Z info: Registered Route: GET /metrics/:metric
server_1   | 2020-05-30T23:35:10.729Z info: Scanning for routes...
server_1   | 2020-05-30T23:35:10.741Z info: Registered Route: GET /user/login
server_1   | 2020-05-30T23:35:10.742Z info: Registered Route: GET /user/login/user/logout
server_1   | 2020-05-30T23:35:10.742Z info: Registered Route: GET /store/order
server_1   | 2020-05-30T23:35:10.742Z info: Registered Route: POST /store/order
server_1   | 2020-05-30T23:35:10.743Z info: Registered Route: GET /store/order/:id
server_1   | 2020-05-30T23:35:10.743Z info: Registered Route: PUT /store/order/:id
server_1   | 2020-05-30T23:35:10.743Z info: Registered Route: DELETE /store/order/:id
server_1   | 2020-05-30T23:35:10.744Z info: Registered Route: GET /pet
server_1   | 2020-05-30T23:35:10.744Z info: Registered Route: POST /pet
server_1   | 2020-05-30T23:35:10.744Z info: Registered Route: DELETE /pet
server_1   | 2020-05-30T23:35:10.744Z info: Registered Route: GET /pet/:id
server_1   | 2020-05-30T23:35:10.744Z info: Registered Route: PUT /pet/:id
server_1   | 2020-05-30T23:35:10.744Z info: Registered Route: DELETE /pet/:id
server_1   | 2020-05-30T23:35:10.745Z info: Registered Route: GET /user
server_1   | 2020-05-30T23:35:10.745Z info: Registered Route: POST /user
server_1   | 2020-05-30T23:35:10.745Z info: Registered Route: DELETE /user
server_1   | 2020-05-30T23:35:10.746Z info: Registered Route: GET /user/:id
server_1   | 2020-05-30T23:35:10.746Z info: Registered Route: PUT /user/:id
server_1   | 2020-05-30T23:35:10.746Z info: Registered Route: DELETE /user/:id
server_1   | 2020-05-30T23:35:10.746Z info: Registered Route: GET /user/count
server_1   | 2020-05-30T23:35:10.746Z info: Initializing routes...
server_1   | 2020-05-30T23:35:10.778Z info: Listening on port 3000...
server_1   | 2020-05-30T23:35:10.784Z info: Starting service MetricsCollector...
```

CHAPTER

SEVEN

PROJECT STRUCTURE

Once you've generated your NodeJS server project with Composer you'll want to familiarize yourself with its structure.



The majority of the important bits are in the `src` folder. So let's focus on that first.

7.1 src/jobs

The `jobs` folder contains TypeScript classes that provide background services that run at a scheduled interval. Each Composer project comes with one default background job called the `MetricsCollector`. The server will automatically pick up any class within this folder and load it as a background job on startup.

7.2 src/models

At startup the server will automatically scan and load all files defined in this folder as data model definitions. Each data model class must be decorated with Composer and TypeORM decorators as Composer uses the TypeORM system to manage storage of all data model objects. Composer will have automatically generated all data models defined in the OpenAPI specification as schemas, adding the desired decorators that were specified in the OpenAPI file.

Each generated model class also features a constructor that takes a single object as its sole argument. This serves as a simple copy constructor which enforces the desired structure of the data model, throwing away any additional undesired data that is passed to it.

7.3 src/routes

The `routes` folder is where most of the magic happens. Each file in this folder is automatically loaded by the server at startup as well. Each file corresponds to one or more Path Item objects in the OpenAPI specification file. When the `x-schema` field is used for a Path Item, then all corresponding path items associated with a given data model are grouped into one class named `<Schema>Route.ts`. If the `x-name` was used then all corresponding path items that specified the same name are grouped into a single class named `<Name>Route.ts`.

7.4 src/config.ts

The `config.ts` file is where all the global application configuration is stored. This contains everything from the background jobs to the datastores and JWT authentication information. The server leverages the nconf configuration system to make defining and accessing configuration variables simple. The resulting `config` object is automatically injected into each **Route** class via the `@Config` decorator.

7.5 src/server.ts

The `server.ts` file is the entry point of the server and is the first script run. It contains simple and common functionality needed to initialize the global configuration, system logger, background service manager and the Composer Server itself.

7.6 test

The `test` folder contains all unit tests for the project. Composer is pre-configured to use the jest unit test framework. Composer automatically generates unit test files for each file in the `src/routes` folder. However, these tests are not complete and must be modified.

CHAPTER

EIGHT

ROUTE HANDLERS

Route Handlers are a collection of ExpressJS middleware functions whose job it is to process an incoming request for a given RESTful endpoint. Collections of handlers are grouped together by a common name or schema, depending on the OpenAPI specification defined.

Composer makes heavy use of both functional and aspect oriented programming techniques. Each Operation defined in the OpenAPI specification is generated to a single function and is decorated with the necessary TypeScript decorators to indicate their function.

8.1 Anatomy of a Route

For example, given the following Operation object defined in an OpenAPI specification file is for a `GET` request to retrieve a single object resource of the `Pet` schema given a specified unique identifier.

```
/pet/{id}:
    x-schema: Pet
    parameters:
        - $ref: "#/components/parameters/id"
    get:
        description: Returns a single Pet from the system that the user has access to
        x-name: findById
        responses:
            "200":
                description: A Pet object.
                content:
                    application/json:
                        schema:
                            $ref: "#/components/schemas/Pet"
```

The resulting route handler class and function as generated by Composer will look like the following.

```
/**
 * Handles all REST API requests for the endpoint `/pet`.
 *
 * @author AcceleratXR, Inc.
 */
@Model(Pet)
@Route("/pet")
class PetRoute extends ModelRoute<Pet> {
        ...
```

(continues on next page)

(continued from previous page)

```
10
11          /**
12           * Returns a single Pet from the system that the user has access to
13           */
14          @Get("/:id")
15          private async findById(@Param("id") id: string, @AuthUser user?: JWTUser):␣
    ↪Promise<Pet | undefined> {
16              return super.doFindById(id, user);
17          }
18
19          ...
20  }
```

First notice how the class itself is constructed. The Path Item in the OpenAPI specification specifies an `x-schema` field with value `Pet`. This means that all Path Items that also specify `Pet` will be grouped into the same `PetRoute.ts` file.

The `@Model(Pet)` decorator is used to identify that this Route handler class is associated with the `Pet` data model. When the Composer server starts up and scans this class it therefore uses this information to bind the correct datastore connection.

The class also uses the `@Route` decorator. This indicates to the Server that the class is responsible for processing RESTful endpoints. The decorator can take either a single string value or an array as its sole parameter corresponding to the root path patterns that the handler will respond to. In this example, this route handler is responsible for processing all incoming requests to the `/pet` path.

Finally we get to the endpoint handler itself, `findById`. The name `findById` directly corresponds to the `x-name` value specified in the Operation object's definition. Further, the HTTP method is denoted by the `@Get` decorator and take an optional argument `/:id`. This path gets appended to the root defined at the top of the class `/pet/:id` when the class is processed by the Server and registered with ExpressJS. Each function defined with a corresponding HTTP method decorator is registered to ExpressJS accordingly as a middleware function. Should the handler also make use of the `@Before` and `@After` decorators, these additional functions will be registered respectively with the route handler function as additional middleware for the given endpoint path.

The function itself takes multiple arguments, namely `@Param("id") id: string, @AuthUser user?: JWTUser`. The First argument with the `@Param` decorator indicates that the path contains one or more parameters, one of which is named `id` and that the server should pull that value out of the request path and pass its value in directly to this argument. Therefore, if the HTTP request that arrives is for the path `/pet/scotty` then the value of the `id` argument will be `scotty`.

The second argument is for the authorized user that performed the request as denoted by the `@AuthUser` decorator. As this is an optional argument, a value is only passed in when a user has actually authenticated with the service and is valid. In all other cases this value will be `undefined` to indicate that no valid user is attempting to perform this action.

The final thing to notice about this function is the body and return type. The return type is of type `Promise<Pet | undefined>`. This indicates to the server that the handler is async and will return a `Pet` object if the object was found with the given id or `undefined` object if it could not be at some point in the future. The return of the function does not have to be a `Promise` and in fact can be a direct value. The server will automatically adjust its behavior accordingly. When one of these values is returned, the server will automatically encode the object as JSON and return it to the client.

The body of the function makes a single call to `super.doFindById(id, user)`. Notice that the class's definition inherits from `ModelRoute`. This is a base class to which all Model route handlers can extend to provide common built-in behavior such as this. It's purpose is to reduce the amount of code needed for common data oriented REST APIs. In this particular case we are leveraging the built-in function `doFindById`. This function performs generic logic for retrieving a single record of the desired type from the datastore. The built-in will also handle validating permissions for accessing the object for the authorized user when Access Control Lists are enabled.

8.2 Route Decorators

The following is a list of decorators that can used within a Route handler class to perform various HTTP processing behavior.

8.2.1 @Route

The `@Route` decorator is used to indicate that a given class contains one or more endpoint handlers for a given set of paths.

```
/**
 * Handles all REST API requests for the endpoint `/pet`.
 *
 * @author <AUTHOR>
 */
@Model(Pet)
@Route("/pet")
class PetRoute extends ModelRoute<Pet> {
}
```

8.2.2 @Init

The `@Init` decorator indicates a function within a Route handler class that should be called at service startup in order to initialize some state.

```
/**
  * Called on server startup to initialize the route with any defaults.
  */
 @Init
 private async initialize() {
     // TODO Add business logic here
 }
```

8.2.3 @Auth

The `@Auth` decorator is used to indicate that the decorated endpoint handle requires authentication by one of the specified methods.

```
 /**
  * Add a new pet to the store
  */
 @Auth(["jwt"])
 @Post()
 private async add(obj: Pet, @AuthUser user?: JWTUser): Promise<Pet> {
     const newObj: Pet = new Pet(obj);

     throw new Error("This route is not implemented.");
 }
```

8.2.4 @Before

This decorator is used to indicate additional middleware functions that should be executed **before** the main endpoint handler is executed. This is typically used for input pre-processing and validation.

```
/**
 * Add a new pet to the store
 */
@Auth(["jwt"])
@Before(["validate"])
@After(["prepareOutput"])
@Post()
private async add(obj: Pet, @AuthUser user?: JWTUser): Promise<Pet> {
    const newObj: Pet = new Pet(obj);

    throw new Error("This route is not implemented.");
}
```

8.2.5 @After

The `@After` decorator is used to indicate additional middleware functions that should be expected **after** the main endpoint handler is executed. This is typically used for post-processing and data preparation before returning to the client.

```
/**
 * Add a new pet to the store
 */
@Auth(["jwt"])
@Before(["validate"])
@After(["prepareOutput"])
@Post()
private async add(obj: Pet, @AuthUser user?: JWTUser): Promise<Pet> {
    const newObj: Pet = new Pet(obj);

    throw new Error("This route is not implemented.");
}
```

8.2.6 @Delete

The `@Delete` decorator is used to indicate that the handler function will process HTTP requests with method `DELETE`. It takes an optional parameter to provide a sub-path.

```
/**
 * Deletes the Pet
 */
@Auth(["jwt"])
@Delete("/:id")
private async delete(@Param("id") id: string, @AuthUser user?: JWTUser): Promise<void> {
    return super.doDelete(id, user);
}
```

8.2.7 @Get

The `@Get` decorator is used to indicate that the handler function will process HTTP requests with method `GET`. It takes an optional parameter to provide a sub-path.

```
/**
 * Multiple Pet objects
 */
@Get()
private async find(@AuthUser user?: JWTUser): Promise<Array<Pet>> {
    throw new Error("This route is not implemented.");
}
```

8.2.8 @Post

The `@Post` decorator is used to indicate that the handler function will process HTTP requests with method `POST`. It takes an optional parameter to provide a sub-path.

```
/**
 * Add a new pet to the store
 */
@Auth(["jwt"])
@Post()
private async add(obj: Pet, @AuthUser user?: JWTUser): Promise<Pet> {
    const newObj: Pet = new Pet(obj);

    throw new Error("This route is not implemented.");
}
```

8.2.9 @Put

The `@Put` decorator is used to indicate that the handler function will process HTTP requests with method `PUT`. It takes an optional parameter to provide a sub-path.

```
/**
 * Updates a single Pet
 */
@Auth(["jwt"])
@Put("/:id")
@Validate("validate")
private async update(@Param("id") id: string, obj: Pet, @AuthUser user?: JWTUser):␣
↪Promise<Pet> {
    const newObj: Pet = new Pet(obj);

    return super.doUpdate(id, newObj, user);
}
```

8.2.10 @Validate

This decorator is used to indicate a middleware function that will execute **before** the main endpoint handler in order to validate the incoming data. It must be the name of a function within the class itself.

```
/**
  * Determines if the specified request payload is valid and can be accepted.
  *
  * @throws When the request payload contains invalid input or data.
  */
 private validate(data: Pet): void {
     // TODO Validate input data
 }

 /**
  * Updates a single Pet
  */
 @Auth(["jwt"])
 @Put("/:id")
 @Validate("validate")
 private async update(@Param("id") id: string, obj: Pet, @AuthUser user?: JWTUser):␣
↪Promise<Pet> {
     const newObj: Pet = new Pet(obj);

     return super.doUpdate(id, newObj, user);
 }
```

8.2.11 @WebSocket

The `@WebSocket` decorator is used to indicate that the handler function will process HTTP Upgrade requests in order to establish a WebSocket connection with the client. It takes an optional parameter to provide a sub-path.

```
 /**
  * Create a WebSocket connection with the client.
  */
 @WebSocket()
 private async connect(@Socket ws: ws, @AuthUser user?: JWTUser): Promise<Array<Pet>> {
     ws.on("message", (msg) => {
         ws.send(`echo ${msg}`);
     });
     ws.send(`hello ${user ? user.uid : "guest"}`);
 }
```

CHAPTER

NINE

BUILT-INS

Built-ins are a way of providing default behavior for common operations in a RESTful API. In the case of Composer, there exists one primary set of built-in functions for working with data models and is provided by the `ModelRoute` abstract base class.

9.1 ModelRoute

The `ModelRoute` base class provides a set of route handler built-ins for common operations when designing data oriented RESTful API services. These built-ins can be called from any route handler function to provide the desired behavior during request processing.

These built-ins are designed around standard CRUD operations and include the following.

- `doCount` - Searches a collection of objects and returns the count of matches
- `doCreate` - Create a new object
- `doDelete` - Delete an existing object
- `doFindAll` - Search and retrieve a collection of objects
- `doFindById` - Retrieve an existing single object
- `doTruncate` - Delete all objects from a collection
- `doUpdate` - Update an existing object

9.1.1 doCount

```
doCount(params: any, query: any, user?: any): Promise<Count>
```

This built-in provides a way to perform a search on a collection of objects and return only the total count of those matching the given criteria. It takes both the request `params` as provided by the `@Param` decorator and the path query parameters as provided by the `@Query` decorator. When ACLs are enabled this built-in will also verify that the authorized user has `READ` permission on the associated type ACL.

9.1.2 doCreate

```
doCreate(obj: T, user?: any, acl?: AccessControlList): Promise<T>
```

This built-in is used to create a single new object and store it in the database. Once stored in the database, the resulting object as it was stored is returned to be delivered back to the client. When ACLs are enabled this built-in will also verify that the authorized user has `CREATE` permission on the associated type ACL. The built-in will also create an ACL for the object, inheriting the permissions of the type ACL and giving full control to the authorized user performing the create. You may optionally provide your own ACL for the object to be created which overrides this default behavior.

9.1.3 doDelete

```
doDelete(id: string, user?: any): Promise<void>
```

The `doDelete` built-in is for deleting a single object in a collection. It takes a single `id` parameter which is the unique identifier of the object in question. The `id` can be any value from an data model class that has the `@Identifier` decorator. This is very useful when data models have multiple possible identifiers such as a `uid` or a unique `name`. When ACLs are enabled this built-in will also verify that the authorized user has `DELETE` permission on the associated object's ACL.

9.1.4 doFindAll

```
doFindAll(params: any, query: any, user?: any): Promise<T[]>
```

The `doFindAll` built-in searches collection of objects for a given set of criteria as specified by the `params` and `query` arguments. Both `params` and `query` arguments must be a map of key=value pairs where the key corresponds to a property within the data model and the value is the desired value to find in the collection. When ACLs are enabled this built-in will also verify that the authorized user has `READ` permission on the associated type ACL.

9.1.5 doFindById

```
doFindById(id: string, user?: any): Promise<T | undefined>
```

This built-in retrieves a single existing object from a collection with a specified unique `id`. The `id` can be any value from an data model class that has the `@Identifier` decorator. This is very useful when data models have multiple possible identifiers such as a `uid` or a unique `name`. When ACLs are enabled this built-in will also verify that the authorized user has `READ` permission on the associated object's ACL.

9.1.6 doTruncate

```
doTruncate(user?: any): Promise<void>
```

This built-in is used to delete all objects of a single collection. It is equivalent to a DROP operation on a table. When ACLs are enabled this built-in will also verify that the authorized user has `DELETE` permission on the associated type ACL.

9.1.7 doUpdate

```
doUpdate(id: string, obj: T, user?: any): Promise<T>
```

The `doUpdate` built-in will modify an existing record in the datastore for the object with the specified id. The `id` can be any value from an data model class that has the `@Identifier` decorator. The `obj` argument is the contents of the new object to persist. It must be the **entire** object and not a partial. The system leverages an optimistic locking mechanism to ensure that only objects with a like `version` can be updated. Thus, if the `version` of the specified object does not match the value of the existing object in the datastore, a `409 CONFLICT` error is thrown and returned to the client. When ACLs are enabled this built-in will also verify that the authorized user has `UPDATE` permission on the associated object's ACL.

CHAPTER

TEN

DEFAULT ROUTES

Composer server projects come with a selection of default routes that are provided out-of-the-box. These include:

- Access Control Lists
- Default Index
- OpenAPI documentation (HTML / json)
- Prometheus metrics

10.1 Access Control Lists

When ACLs are enabled (via `config.ts`) the server automatically registers the `/acls` route handler. This route handler responds to the standard built-in operations for the endpoints for all requests sent to the `/acls` path as follows.

- `GET /acls` - Searches all ACLs
- `POST /acls` - Creates a new ACL
- `DELETE /acls` - Deletes all ACLs
- `GET /acls/count` - Searches all ACLs and return the count
- `GET /acls/:id` - Returns a single existing ACL
- `PUT /acls/:id` - Updates a single existing ACL
- `DELETE /acls/:id` - Deletes a single existing ACL

When the server starts up, the system automatically scans all route handler classes that inherit from `ModelRoute` and retrieve their `defaultACL`. This default ACL is associated with all collection oriented requests for a given data model (e.g. `create`, `findAll`, `truncate`) where the `uid` of the ACL is the name of the data model class.

Individual record ACL objects are associated with their actual objects by using the `uid` value for both the object and the ACL itself. These objects will then specify the type ACL as it's parent, in order to inherit default behavior from the class.

10.2 Default Index

The default index route handler binds to the / endpoint and handles all GET requests. It is a simple handler that returns basic information about the service such as its name, version and server time. The resulting output looks like this.

{"name":"petstore","time":"2020-05-29T21:09:36.123Z","version":"1.0.0"}

10.3 OpenAPI Documentation

The OpenAPI default route provides access to the server's OpenAPI specification in both HTML and json form.

- GET /api-docs - Returns the server's OpenAPI specification in HTML format
- GET /openapi.json - Returns the server's OpenAPI specification in JSON format

10.4 Prometheus Metrics

The Composer server comes with built-in support for Prometheus metrics and is exposed via the /metrics endpoint.

- GET /metrics - Returns all Prometheus metrics for the server
- GET /metrics/:name - Returns the Prometheus metric with the given name

CHAPTER

ELEVEN

DEBUGGING

While it is perfectly possible to run the command `yarn start` to get the server going, this also requires that you've got a database up and running locally as well. This is obviously not ideal so Composer comes pre-packaged with configuration for Docker. With Docker you can run the service in a container, including all required databases from the command prompt.

11.1 Docker Compose

```
docker-compose build
docker-compose run
```

11.2 Debugging in Visual Studio Code

Composer also is designed for work with Visual Studio Code out of the box. If you are using VS Code all you need to do is hit `F5` once your docker container is up and running. It will automatically connect to your instance and allow for debugging of the service directly.

Composer is a simple, light-weight and opinionated framework for rapidly developing scalable REST API services for NodeJS, written in TypeScript.

The framework combines the OpenAPI specification with a simple functional programming model to provide a highly expressive and powerful system for developing REST API services.

CHAPTER

TWELVE

FEATURES

- Built on OpenAPI
- HTTP 1.x/2.x Web Server
- WebSocket Support
- Global Configuration
- Dependency Injection
- Built-in behaviors for common REST API actions
- Built-in ORM layer
- Document Version Tracking
- MongoDB support
- Redis support
- 2nd Level Caching
- Authentication
- Role Based Access Control
- Prometheseus metrics

CHAPTER

THIRTEEN

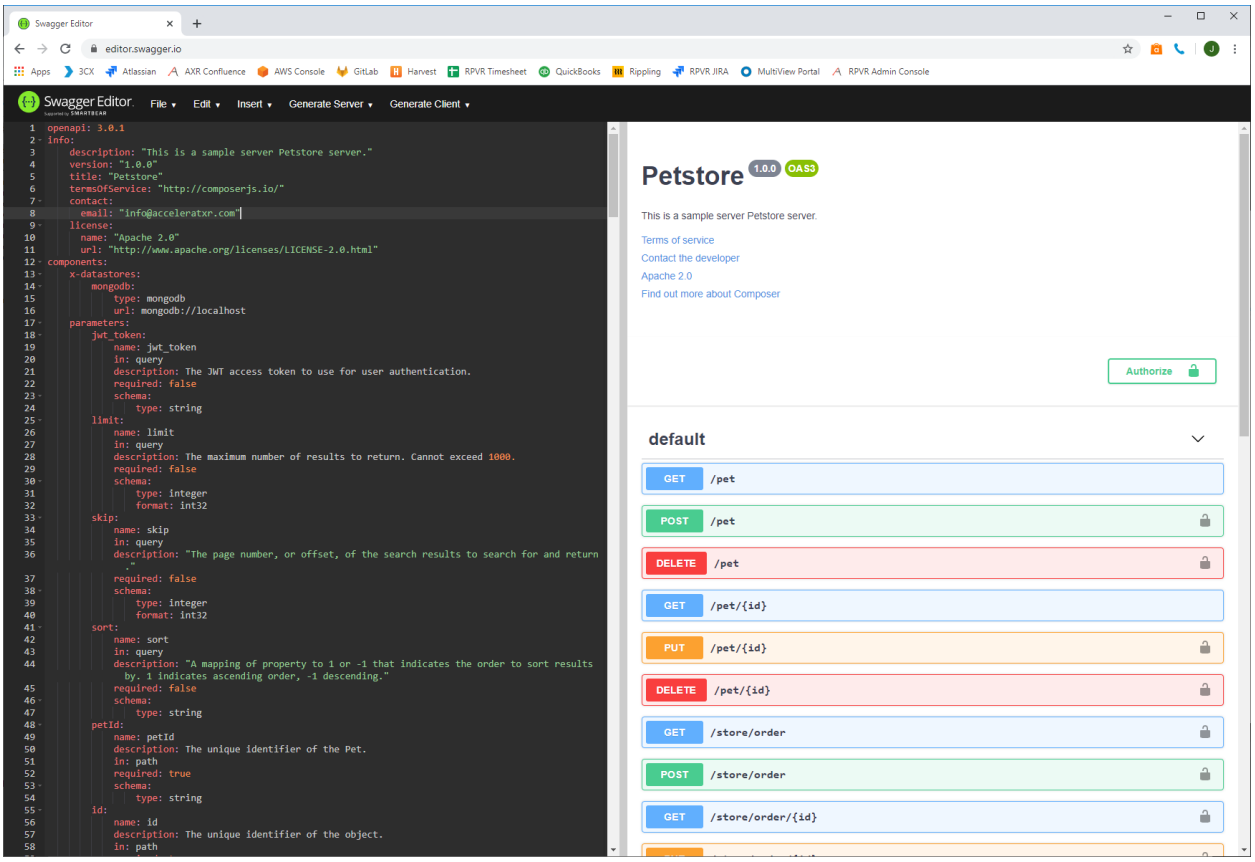
GETTING STARTED

Using Composer is as simple as 1-2-3.

13.1 Step 1

Define the REST API service using OpenAPI and save it as `service.yaml`.

Click here for an example.



13.2 Step 2

Generate the project using the Composer CLI.

```
yarn global add @composer-js/cli
composer -i ./service.yaml -o . -t server -l nodejs
```

13.3 Step 3

Start the server!

```
yarn install
yarn start
```