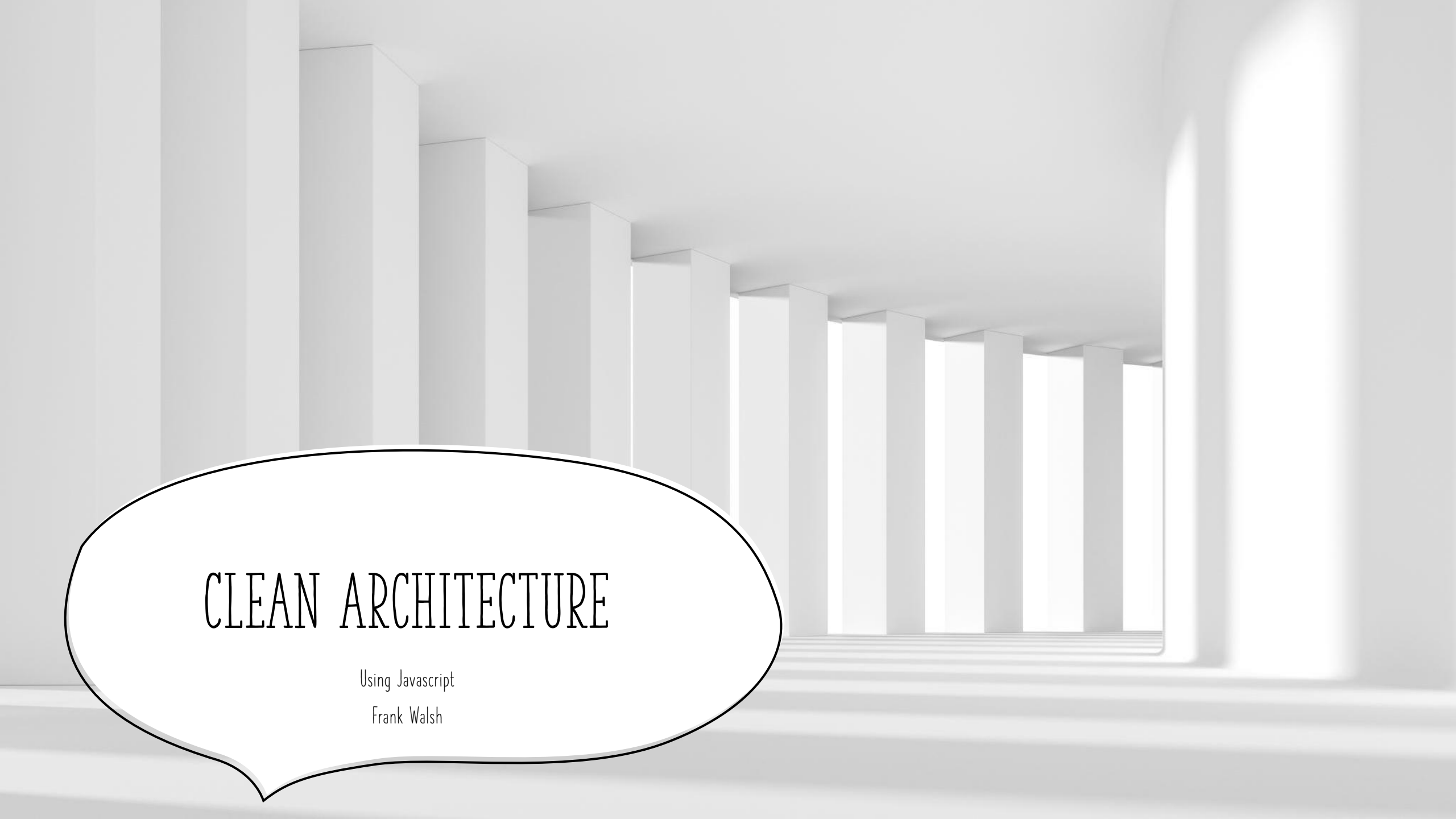


CLEAN ARCHITECTURE

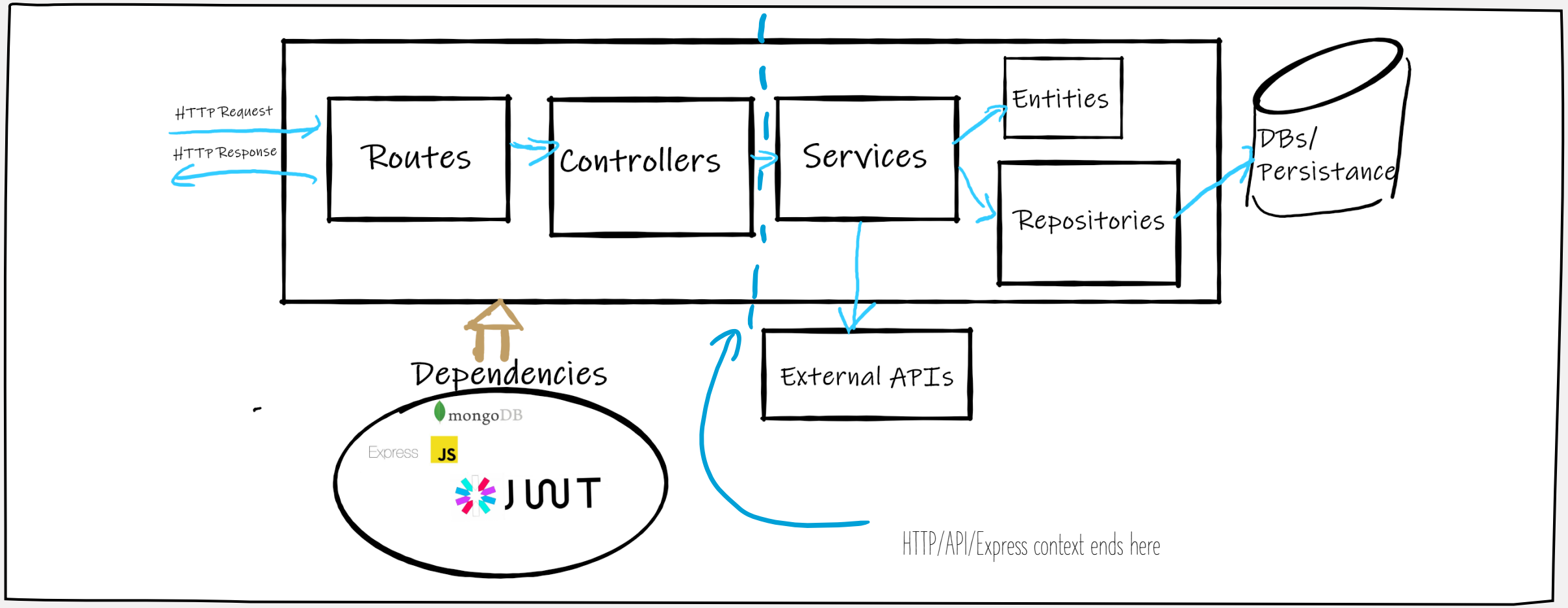
Using Javascript

Frank Walsh

WHAT'S CLEAN ARCHITECTURE

- Lots of different architectures/project Structures
  - Clean, Model/View/Controller, Domain Driven Dev.
- Clean Architecture is a software design pattern for building scalable and maintainable systems
- Key concepts for Clean Architecture
  - Separation of Concerns(SoC): *dividing the application into four concentric layers, with each layer having a specific role*
  - The Dependency Rule: *code dependencies must always point inwards*

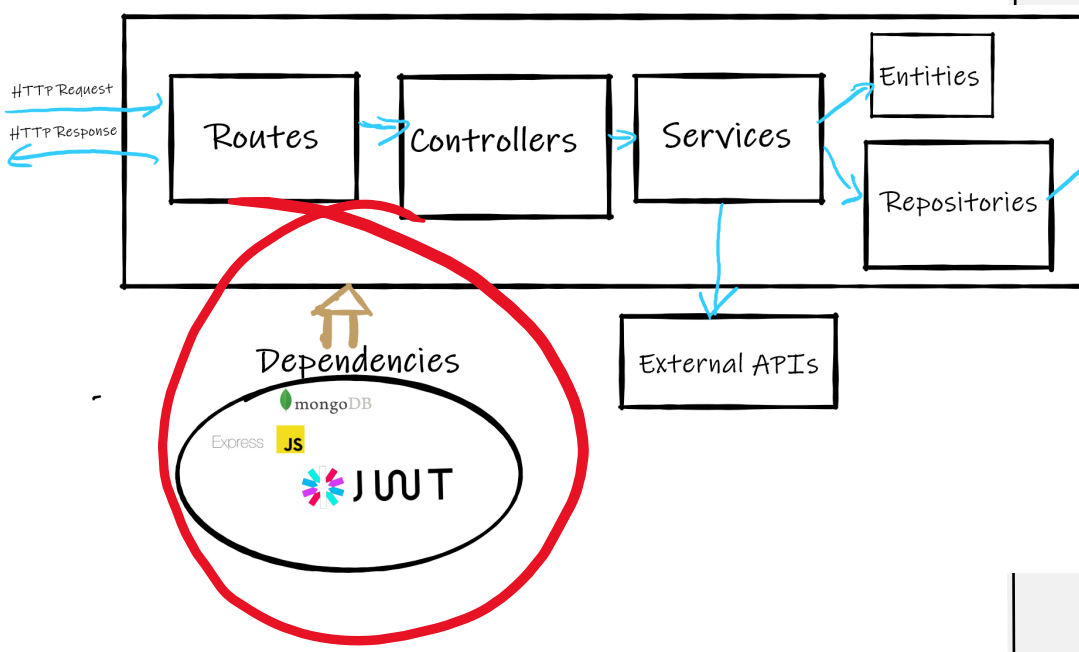
See the next slide....



A CLEAN ARCHITECTURE INSPIRED APPROACH

DEPENDENCY INVERSION/DEPENDENCY INJECTION

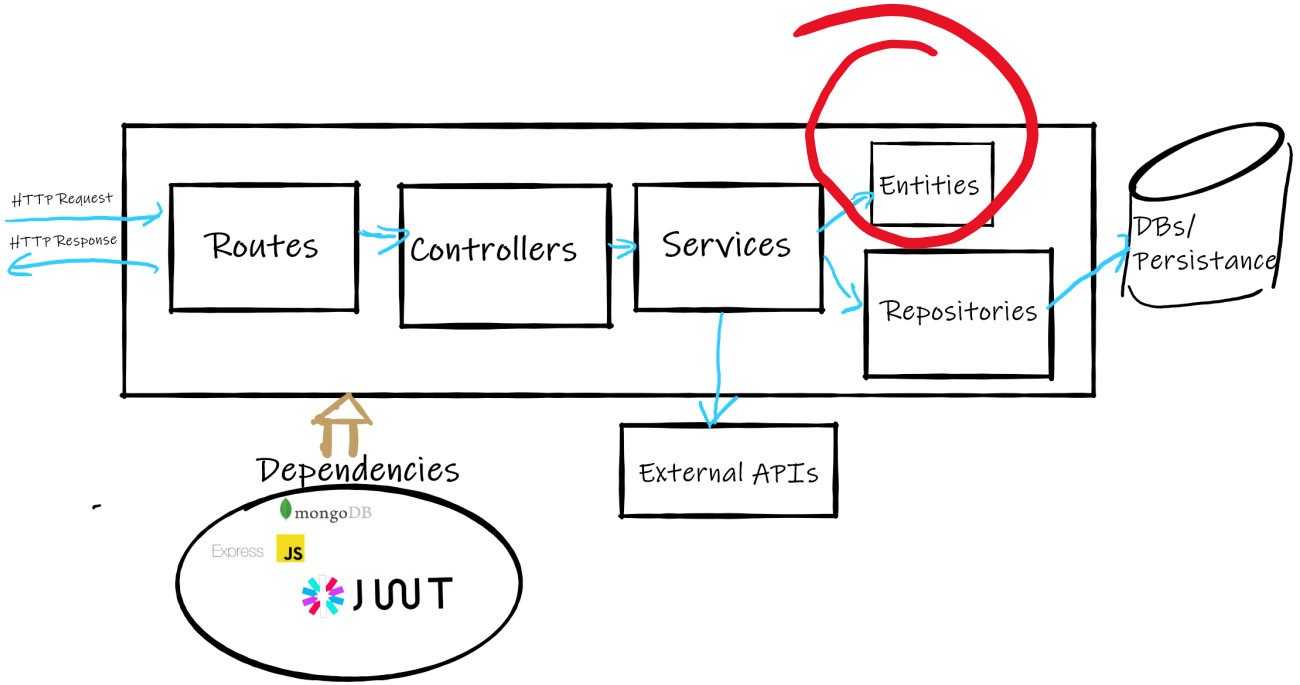
- Dependency Injection (DI) can be used to manage dependencies between components, ensuring that the core business logic remains decoupled from external dependencies.
- Define abstractions (interfaces) for dependencies
  - E.G. define interface that represent the abstract operations required for data persistence
  - In the outer layers, create concrete implementations of the interface defined in the inner layers
  - Inject dependencies: When instantiating components, inject the concrete implementations of the required interfaces as constructor or function parameters



ENTITIES

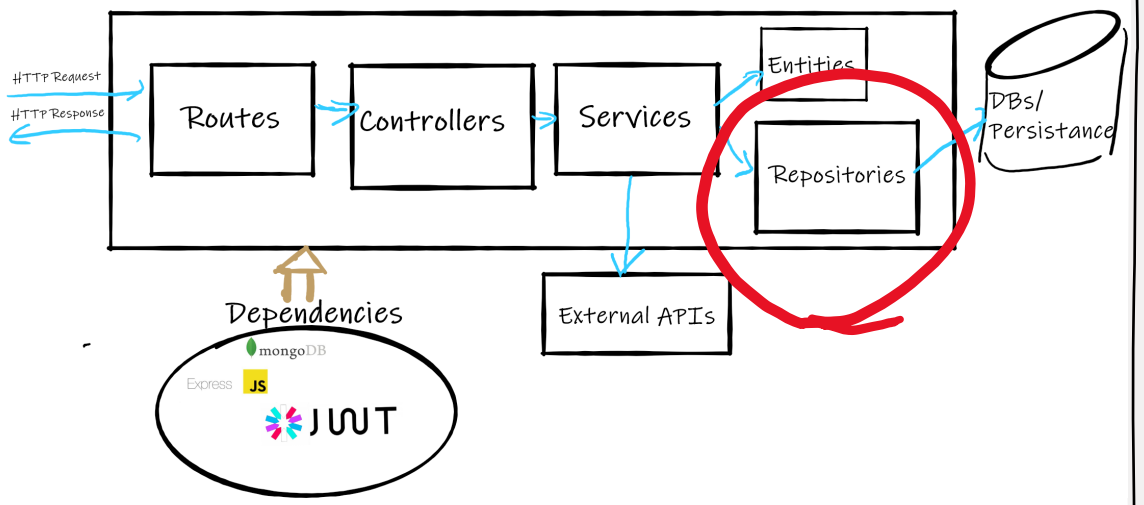
- Represent the core business objects
  - E.g. "Movie","User","Account"

```
export default class User {
    constructor(id, name, email) {
      this.id = id;
      this.name = name;
      this.email = email;
    }
  }
```



REPOSITORIES

- Defines the Database/Data Store Interactions
- Define "Interface Description" for Repository
- Provide implementation



UserRepository.js

```
export default  class {

  persist(account) {
    throw new Error('ERR_METHOD_NOT_IMPLEMENTED');
  }

  merge(account) {
    throw new Error('ERR_METHOD_NOT_IMPLEMENTED');
  }

  remove(accountId) {
    throw new Error('ERR_METHOD_NOT_IMPLEMENTED');
```

Implementation extends "Interface Description"

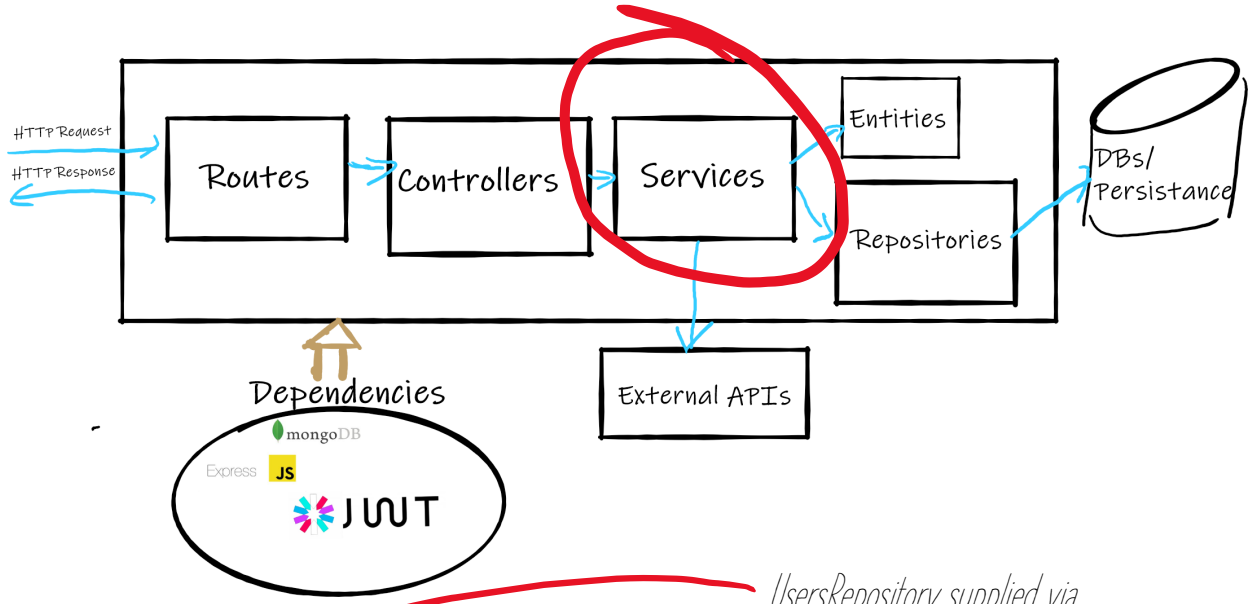
In future, will implement for MongoDB

InMemoryRepository.js

```
export default class extends UserRepository {

  dataAsArray() {
    return Object.keys(this.data).map(key => this.data[key]);
  }

  constructor() {
    super();
    this.index = 1;
    this.data = {};
  }

  persist(accountEntity) {
    const row = Object.assign({}, accountEntity);
    const rowId = this.index++;
    row.id = rowId;
```

SERVICES

- Contains the applications business logic that our API supports(e.g. register new user, get movie details).



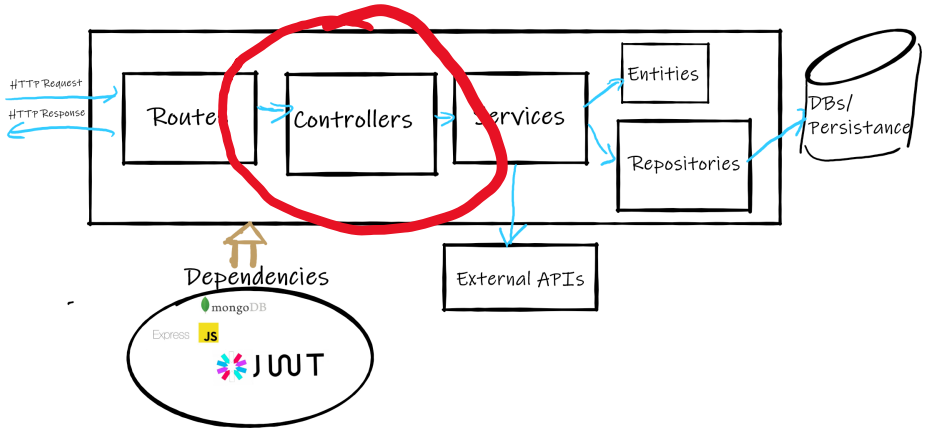
UserService.js

```
import User from '../entities/User';

export default {
  registerUser: async (name, email, { usersRepository }) => {
    const user = new User(undefined, name, email);
    return usersRepository.persist(user);
  },
  find: ({ usersRepository }) => {
    return usersRepository.find();
  }
};
```

UsersRepository supplied via Dependency Injection

CONTROLLERS

- For API, the Controllers extract the parameters (query and/or body) from the HTTP request, call the relevant service, and return the HTTP response.

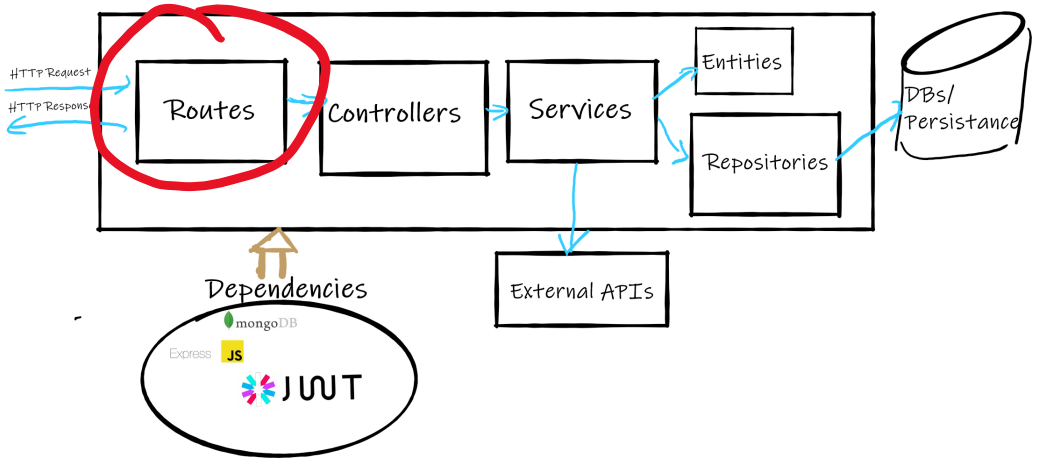


Object containing External Dependencies supplied as argument to the function that returns the controller

```
import userService from "../services/userService";

export default (dependencies) => {

    const createUser = async (request, response, next) => {
        // Input
        const { name, email } = request.body;
        // Treatment
        const user = await userService.registerUser(name, email, dependencies);
        //output
        response.status(201).json(user)
    };
    const listUsers = async (request, response, next) => {···
    };

    return {
        createUser,
        listUsers
    };
```

ROUTES

- Roouters handle the HTTP requests that hits the API and route them to appropriate controller
- They can also be used to chain together several controllers.
- Connects the Express.js Framework to the architecture




```
import express from 'express';
import userController from '../controllers/userController';

const createRouter = (dependencies) => {
    const router = express.Router();
    // load controller with dependencies
    const controller = userController(dependencies);
    router.route('/')
        .post(controller.createUser);

    router.route('/')
        .get(controller.listUsers);

    return router;
};
export default createRouter;
```

EXTERNAL DEPENDENCIES - DATABASE

- A component that exists outside of the core business logic of an application. :
  - E.g. databases, user interfaces, third-party libraries/frameworks
- In this example, the dependencies.js module exports a function that builds a dependency object.
- Use it to "inject" infrastructure dependencies(such as Repositories) into the routes/services.
- For now, we just have DB details

```
import InMemoryRepository from '../repositories/InMemoryRepository';

const buildDependencies = () => {
  const dependencies = {
  };

  if (process.env.DATABASE_DIALECT === "in-memory") {
    dependencies.usersRepository = new InMemoryRepository();
  } else if (process.env.DATABASE_DIALECT === "mongo") {
    throw new Error('Add Mongo Support');
  } else if (process.env.DATABASE_DIALECT === "mysql") {
    throw new Error('Add MySQL support');
  } else {
    throw new Error('Add DB Support to project');
  }
  return dependencies;
};

export default buildDependencies;
```

BRINGING IT TOGETHER! - THE API "ENTRY POINT" INDEX.JS

```
// Load the http module to create an http server.
import createUsersRouter from './src/routes/userRouter';
import buildDependencies from "./src/config/dependencies";
import express from 'express';
import dotenv from 'dotenv';

dotenv.config()
const port = process.env.PORT
const dependencies = buildDependencies();

const app = express();
app.use(express.json())

app.use('/api/users', createUsersRouter(dependencies));

app.listen(port, () => {
  console.info(`Server running at ${port}`);
});
```

Import functions that will build router and dependencies for API

Build dependencies object for API

Create Router and "inject" dependencies