# Global Error Handling Technical Design

## Error Handling

### Functional Requirements

* When an error occurs, an error dialog must be displayed to the end user
* In a Production environment, the error must be displayed in a friendly message. No implementation details should be shown to the end-user.
* In a Development environment, the error message must contain rich detailed information in order to support testing, development, and debugging activities.
* All errors must be logged to a persistent store (i.e. database)

### Design Goals

* **Reusability** – the global error handling services and components will be implemented as a reusable module so that it can be leveraged by any Angular application
* **Flexibility** – the design will be flexible with respect to how errors of different types can be transformed to a common error structure so that error information can be consistently captured and easily logged
* **Configurability** – The error dialog should be configurable so that an error dialog will show rich detailed error information in a development environment; and show a general friendly error message in a production environment.
* **Reliability and Resiliency** – For web service errors, there should be a way to specify retries (and the retry count) at the point that the service call is made.
  + Retries should be disabled by default.
  + The error should be re-thrown if the maximum retry count has been reached.
  + Errors should propagate to the Global Error Handler.
* **Modularity** – Modules, components, and services will be loosely-coupled.

### Design Patterns

* **Singleton** – GlobalErrorHandler will be an application-wide singleton
* **Dependency Injection** – All services will be register in a DI container to decouple service dependencies
* **Middleware** – The retry handler will be implemented as an HTTP request interceptor so that it can be leveraged by all HTTP requests.
* **MVC** – The UI components will follow the MVC architecture.

### Architectural Principles

* Reusability
* T-DRY – Try to be DRY. Do not Repeat Yourself
* KISS – Keep it Simple, Stupid.
* LIFT – Locate, Identify, Find, and T-DRY for folder structure
* SRP – Single Responsibility Principle

### Design Constraints

* All errors are handled by a catch-all global error handler
* All errors caught by the global error handler will be logged to the database via a web service call

## Wait Handler

### Functional Requirements

* For long running HTTP requests, there should be a way to display a wait indicator
* There are two types of wait indicators – a spinner and a wait dialog

### Design Goals

* **Reusability** – the wait indicator feature will be implemented as a reusable module so that it can be leveraged by any Angular application
* **Flexibility** – the design will be able to accommodate two types of wait indicators: a spinner or a wait dialog.
* **Configurability** – clients of the module will able to choose between showing a spinner or a wait dialog.
* **Reliability and Resiliency** – the wait indicator will work even when errors occur.
* **Modularity** – Modules, components, and services will be loosely-coupled.

### Design Constraints

* Clients of the module will not be able to customize the spinner or the wait dialog.

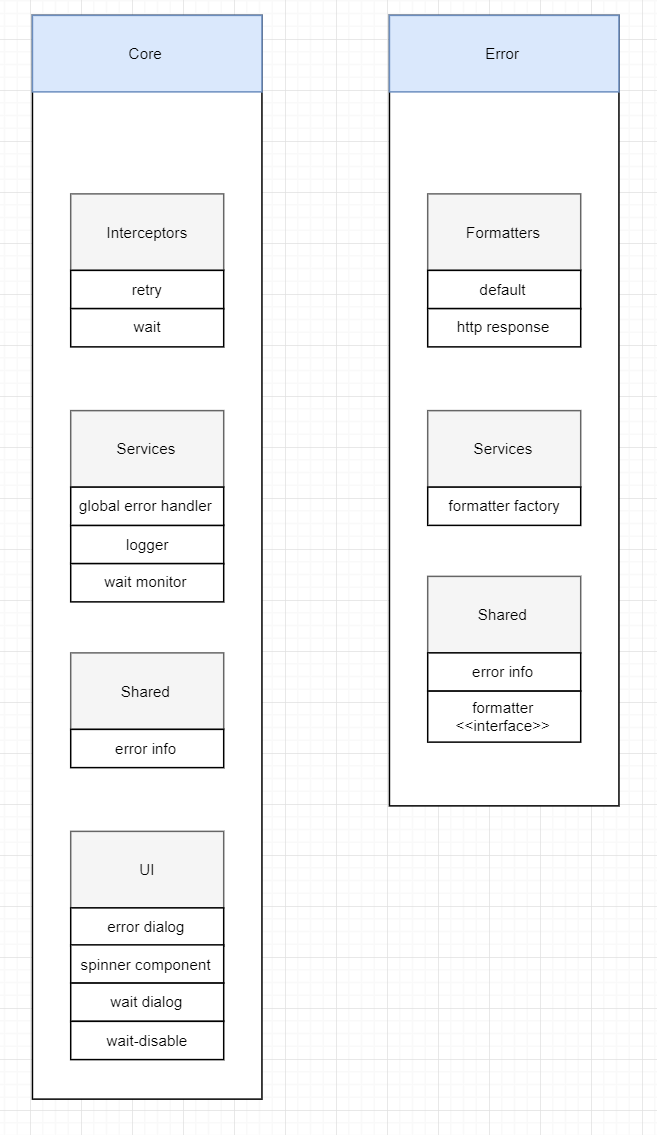
### Design Patterns

* **Singleton** – WaitMonitor will be an application-wide singleton
* **Dependency Injection** – All services will be register in a DI container to decouple service dependencies.
* **Middleware** – The wait handler will be implemented as an HTTP request interceptor so that it can be leveraged by all HTTP requests.
* **MVC** – The UI components will follow the MVC architecture.

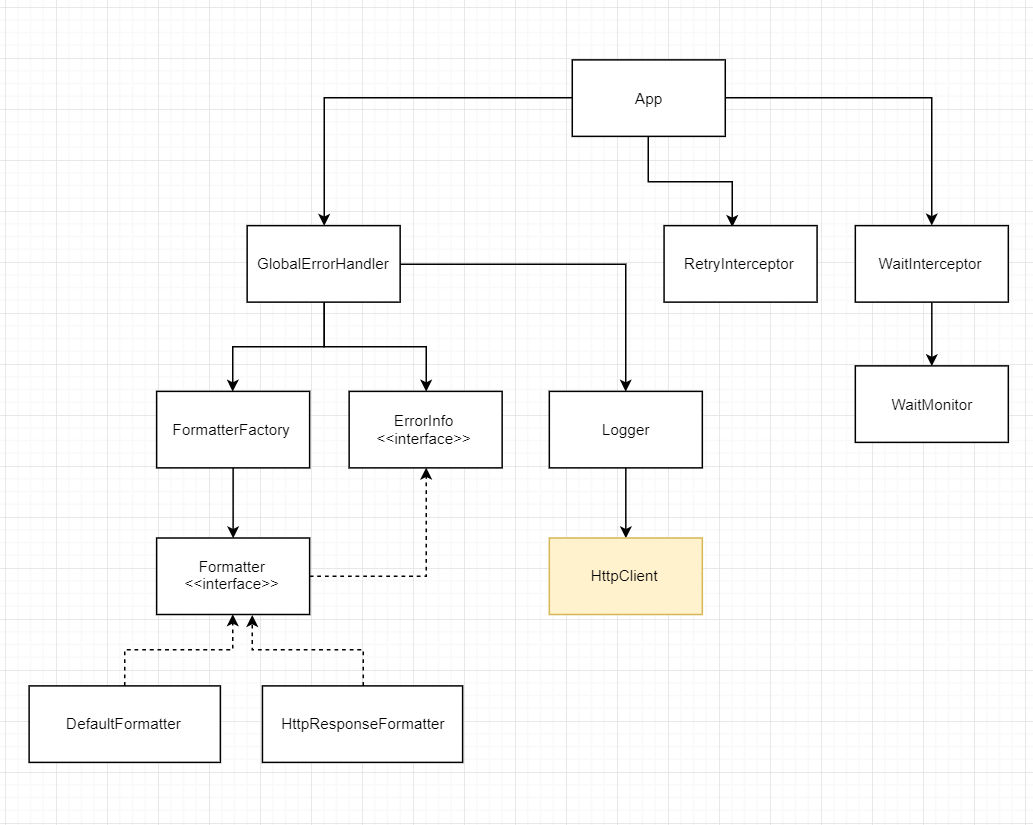
### Architectural Principles

* Reusability
* T-DRY – Try to be DRY. Do not Repeat Yourself
* KISS – Keep it Simple, Stupid.
* LIFT – Locate, Identify, Find, and T-DRY for folder structure
* SRP – Single Responsibility Principle

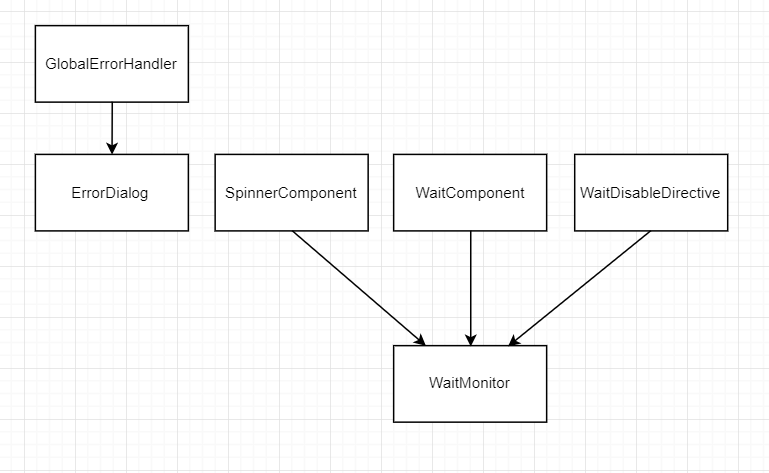
# Package Diagram



# Service Dependency Diagram



# Component Diagram



# Module Dependency Diagram

