Table of Contents

[Front end 2](#_Toc40969332)

[Models 2](#_Toc40969333)

[Components 2](#_Toc40969334)

[Creating a Component (With CLI) 2](#_Toc40969335)

[Create a Component Within a Folder 3](#_Toc40969336)

[General Life Cycle of Components 4](#_Toc40969337)

[Services 4](#_Toc40969338)

[Assessment Service 4](#_Toc40969339)

[AssessmentEngine Service 4](#_Toc40969340)

[Attachment Service 5](#_Toc40969341)

[Helper Service 5](#_Toc40969342)

[Login Service 5](#_Toc40969343)

[Question Service 5](#_Toc40969344)

[Validation Service 6](#_Toc40969345)

[Back End 6](#_Toc40969346)

[General Server Information 6](#_Toc40969347)

[Items Installed on the Server 6](#_Toc40969348)

[Run the Server (locally) 7](#_Toc40969349)

[Development Port (Testing and Viewing Responses) 7](#_Toc40969350)

[How it Works 7](#_Toc40969351)

[Connecting to the Server from Within Express 7](#_Toc40969352)

[Connection Strings 7](#_Toc40969353)

[HTTP vs HTTPS 7](#_Toc40969354)

[API (app.js) 7](#_Toc40969355)

[Email.js 8](#_Toc40969356)

[File-engine (engine.js, check-upload-answer.js) 8](#_Toc40969357)

[Question Factory 8](#_Toc40969358)

[Login and Authorization (routes/user.js and middleware/check-auth.js) 8](#_Toc40969359)

# Front end

The front end of the application uses Angular to serve up content by making asynchronous calls to our API. All navigation is done by initializing and destroying components, rather then navigating to different URLs. Additionally, services are refreshed when needed to make sure that values do not carry into the next view. All of the data on the front end is driven by models. This data is passed into or sent out of components. Finally, services are used to communicate and make requests to the back end. Services are also used to temporarily store data and hold common functions.

## Models

The models folder holds all of the class definitions of the data objects in the application. All question-types models implement the Question interface. Throughout the application, these question objects are passed around generically as a Question. They are then cast to a specific **questionType** when a specific action is being taken, such as validating or grading a question object.

With Angular, we are able to use these objects to both calculate and store values with a code behind (.ts) file, as well as display and update values in the component views (.html).

Many models import other models to include in their structure. The Checkbox Model, for example, imports the Option, Attachment, and Category models to use when building out its own model.

These front end models vary slightly from their back end counterparts. A factory is used on the back end to transfer data between the two models.

## Components

The components in this application are standard Angular components. They consist of a view (html) file, a style (css) file, and a code-behind (ts) file. Most components are broken down into four separate components: create, edit, view, and list. The ability to delete objects is usually found within edit and list components. These component sets are placed into folders and grouped together in similarity to make it easier to understand and for styling purposes.

For example, if you navigate to src > app > questions > questionTypes, you will see a folder for each question type in the application. You will also see a \_taskStyles folder. This folder holds three styling files: createStyles, editStyles, and viewStyles. Each **questionType** *create* component shares the createStyles file, and so forth.

### Creating a Component (With CLI)

Open a new terminal and make sure that you are inside of the project.  
The following syntaxes will generate a component:  
**ng generate component <ComponentName>  
ng g c <ComponentName>**

#### Create a Component Example

ng generate component ViewAssessment  
or  
ng g c ViewAssessment

**Results:**

* Files will be created with view-assessment as the base of the name for each file.
  + view-assessment.component.css
  + view-assessment.component.html
  + view-assessment.component.spec.ts
  + view-assessment.component.ts
* The name of the component will be ViewAssessmentComponent
* Component is added as an import to the app module
* Component is added to the declarations array of the NgModule object within the app module

### Create a Component Within a Folder

To create a component with a folder, simply modify the command to be folderName/ComponentName

**Examples:**

ng generate component assessments/ViewAssessment  
or  
ng g c assessments/ViewAssessment

**Results:**

* Create a new set of component files inside of the assessments folder
* Files will be created with view-assessment as the base of the name for each file.
  + view-assessment.component.css
  + view-assessment.component.html
  + view-assessment.component.spec.ts
  + view-assessment.component.ts
* If the assessments folder did not exist, the command would create the folder as well
* The name of the component will be ViewAssessmentComponent
* Component is added as an import to the app module
* Component is added to the declarations array of the NgModule object within the app module

**Important!**If you make a mistake when creating a component and you want to remove it, you must remove the import and the declarations array from the app module, along with all four files that make up the component.

### General Life Cycle of Components

Most components in the application follow a similar life cycle:

**ngOnInit()**

This function is called when the component is being initialized. It is often used to make calls to the API to fetch data for the view. It is also used to send data off to a service so the data can be worked with as the user interacts with the screen.

**onSubmit()**

A lot of the components in this application have a form built into them. onSubmit() is used throughout the application to handle the data that the user has entered into the inputs on the screen. This function is often holding calls to services to validate data, as well as a call to the API to save or update data in the database.

**onDestroy()**

Because this is a Single Page Application, there is a bit of cleanup work that needs to be done when a component is no longer needed. This function is used to unsubscribe listeners and to reset service values, so the service is ready to work with a fresh component.

## Services

The services in this application serve three main purposes. First, they hold data that is being shared by multiple components. When this data is updated, the service pushes out events to the subscribers that are found in the components. The second purpose of services is to make the http requests to the back end API. All http requests in this application are delegated to services. In this way, all of the properties for loading states and success messages can be handled in a more abstract way. Finally, services hold common functions that are reused throughout the application. The functions in the ValidationService, for example, are reused for every component that includes a form. There are different services for different areas of the application.

### Assessment Service

The assessment service is used with components that handle creating, editing, deleting, and viewing assessments from the instructor’s point-of-view. There are also functions to handle the AssessmentConfig CRUD operations because AssessmentConfig objects are nested in Assessment objects.

### AssessmentEngine Service

This service is used for when a student is taking an assessment. This starts with logic to generate an assessment URL by creating a shell TakenAssessment object to save to the database, then building out a URL with that new objectId. The AssessmentEngine Service holds functions to prepare and start up an assessment. It also holds functions for grading a question immediately after it is submitted. The results of the auto grading, as well as the assessment configuration settings, drive the logic as to what is done next. Eventually, all logic paths lead to the assessment being submitted, which is done with an http request to update the TakenAssessment object.

The only question type that is not graded on the front end is the Upload question. The files that the student uploads are added to the question object, and the whole question object is passed to the back end. The file-compare engine extracts (if necessary) and compares the contents of the files uploaded by the student to the files stored as the correct answer by the instructor using a recursive pattern. A Boolean value is then passed to back to the front end. The AssessmentEngine awaits this result from the server before it decides which path to take next in the algorithm.

Finally, the code for saving and updating the Student record is in the AssessmentEngine. These http requests are kept here because the Student record is first created and saved when the assessment begins. Additionally, any updates that happen to a student record is done so after searching for TakenAssessment records, which is also managed by the AssessmentEngine.

### Attachment Service

There are three different sets of attachments that are managed in this service. The first, is simply called attachments. This is an array of files that an instructor uploads to a question for the student to view. If it is an image, there is logic to display it immediately on the screen. If it is anything else, a download link is provided from this service.

The second set of attachments that is managed in the Attachment service is the correctAnswers. This array of files holds what will later be used as the correct answer for an Upload question. Components that allow CRUD operations for Upload questions will utilize this array.

The final set of attachments is the submittedAnswers array. This array stores the files that the student uploads while taking an assessment.

While this service is typically only handling the attachments array, a single Upload question might be using all three sets of attachments at the same time.

### Helper Service

This service holds reusable functions that are common amongst most components. The openSnackBar() function for example is written with a set of parameters that allows it to be used any time a message needs to appear for a user. Conversion functions and some refresh component functions are also found here.

### Login Service

This service holds logic that is specific to logging in and out of the application. It also contains logic for creating a new user, including checks to see if the user already exists. Auth data logic and tokenization logic are also stored here.

### Question Service

Easily the largest service in the application, the Question service handles all CRUD operations for all question types. It also handles CRUD operations for Options, Categories, and ExactMatches because they are all so closely related to questions. This service is reset after each operation to prevent values from persisting when the user is working with a new question.

### Validation Service

This service handles all form validation for the entire application. All regEx expressions are found in this service, and all forms run checks through this service before making calls to a different service to perform a CRUD operation.

# Back End

The backend holds the API of the application. While it is mostly used for communicating with the database, the file-engine section is used for auto-grading Upload questions during an assessment. Additionally, the login and authorization logic is separated from the rest of the api calls to make the structure of the application easier to follow.

## General Server Information

### Items Installed on the Server

* NodeJs
* MongoDb
* Iisnode
* UrlRewrite
* Chrome
* VS Code
* CertifyTheWeb

#### In NodeServer folder > node modules

* npm install express
* npm install mongoose
* npm install adm-zip
* npm install bcrypt
* npm install jsonwebtoken
* npm install nodemailer
* npm install copy-to-clipboard

### Run the Server (locally)

Open a new terminal window and enter the following command:  
**npm run start:server**

### Development Port (Testing and Viewing Responses)

The response can be seen by opening a browser and navigating to the following URL:  
<http://localhost:3000/>

For example, <http://localhost:3000/api/questions> will make a GET request for all questions.

### How it Works

The server file (server.js) creates the server, then makes an open ended call to listen for requests. We are using the HTTP module that is built-in to Node.js to transfer the data (requests and responses). Additionally, there is some error handling with specific messages for permissions and the port already being used.

When we make the call to create the server, we pass through our Express app as a parameter. The Express app (backend/app.js) holds our API library, as well as permissions and other server-side logic.

### Connecting to the Server from Within Express

Method with connection string is located in Express app (backend/app.js)  
User: expressApp  
Password: Ohi6uDbGMZLBt56X

### Connection Strings

**Development/GitHub:**   
mongodb+srv://expressApp:Ohi6uDbGMZLBt56X@cluster0-bomls.mongodb.net/bitsDemo?retryWrites=true&w=majority

**Server:**  
mongodb://localhost:27017/BITS?retryWrites=true&w=majority

### HTTP vs HTTPS

Everything done in development is done in http. In the production environment, all calls are automatically rerouted to use https.

## API (app.js)

The top of this file contains the connection string to the database, as well as the CORS headers options that are used for the entire API. The bulk of the API for the application is found in the app.js file. With the exception of file-compare and login logic, all http requests that hit the server flow through this file. Error handling for this API occurs in two ways. First, a 400 series response object is sent back to the client with a custom error message. Second, a more detailed error message is logged on the server for developers to see. This log is stored on the server at C:\NodeServer\iisnode. It contains an index.html for viewing the logs in a table view. It also contains .txt files of the error logs.

## Email.js

This file is used for sending emails of the assessment results upon the TakenAssessment object being successfully stored in the database. The logic in this file includes functions to create the body of the email. It also has some formatting functions to convert values into a more readable format. Where the email is going is dependent on the campusLocation value that is passed in via the Student object, which is nested in the TakenAssessment object. There is a constants.js file that holds key/value pairs for the “to” email address. The key is the campus location, and the value is the email address to send it to.

## File-engine (engine.js, check-upload-answer.js)

The file-engine folder contains engine.js, which is a library of functions that was built out to handle file comparing within the application. When the front end makes a call to the back end to do a file compare for an Upload question, the question object is passed to check-upload-answer.js. This file contains an algorithm that utilizes the engine.js library to extract and compare the contents of the student submitted files to the correct files that were uploaded by the instructor. The compare contents logic itself utilizes the recursive pattern to continue to drill down and extract if necessary until all sub-directories have been ran searched through. It then reads and compares the contents of the files on the way back up.

## Question Factory

The Question factory is used on the backend to convert the frontend objects into objects that store into the database. Although the application is utilizing a non-relational database (MongoDb), it is still using logic in the back end that uses relationships to validate data before saving it. When it is updating questions, it converts all values to an anonymous object with property/value for each value that needs to be updated. The object is then passed into an update request with Mongoose.

## Login and Authorization (routes/user.js and middleware/check-auth.js)

This user.js file is used to create, find, and log in users. When a user attempts to log in, first a request is made to find the username in the database. If a user is found, then the password entered by the user is hashed and compared to the hashed password saved in the database. If the passwords match, a token is created which contains a secret and sets an expiration of 2 hours (in string notation). The token information (with expiration converted to seconds) is then passed to the front end along with the user’s isAdmin status.

The check-auth.js file is a middleware that runs on all requests to the backend, except for those related to taking a test and logging in. It exports an anonymous middleware which compares the token of the current user with the secret. If the secrets do not match, then authorization fails. If authorization fails, the data is blocked from being retrieved or pushed to the database.