RESTful Web Services

Accessing Server Data through RESTful Web Services

Introduction

Previous assignments described using HTML, React and Redux to build a dynamic single page application (SPA) as a prototype of a course manager application called WhiteBoard. The SPA was implemented entirely as a client side application running entirely on a browser. This assignment expands on SPA to provide server side support and state. Feel free to reuse the HTML, CSS, React and Redux implemented used in previous assignments. The SPA will interact with RESTful services implemented using Spring Boot.

Learning objectives

The learning objectives of this assignment are

- Implement RESTful Web services
- Implement a data model using Java
- Expose a data model through a REST API
- Retrieve data from REST Web services
- Post data to REST Web services
- Update and delete data from REST Web services

Create a Spring Boot Java server application

If you have not done so already, create a spring boot java server application using the spring command line as follows:

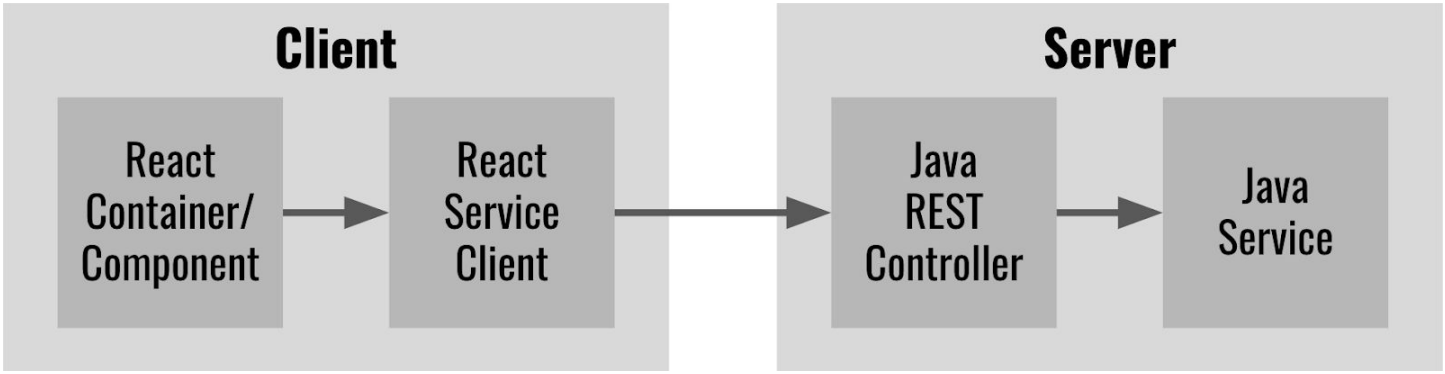
spring init whiteboard --dependencies=web

Import the maven project generated into your favorite IDE

Implement Widget RESTful service (50pts)

As the user interacts with the user interface (UI) on the browser, the React.js application modifies the DOM to reflect the changes. Changes in the client are temporary and can only be made permanent with the help of a server. Clients can notify servers of changes in the state of the application through the use of HTTP requests to specific URL endpoints published by the server through a technology called AJAX (Asynchronous JavaScript and XML). We refer to these endpoints as Web services. RESTful Web services impose certain conventions on the syntax and methods of the HTTP URLs used to communicate with the Web services. The conventions are based on best practices applicable when the end points are used to manipulate some data model on the server. Data models usually provide create, read, update, and delete operations (CRUD) operations. RESTful Web service conventions state that URLs should be nouns describing the data model, e.g., /widgets, /courses, /modules, etc. REST conventions also state that HTTP POST will be used to create,

DELETE to remove, GET to read, and PUT to update instances of the data model. Create a RESTful Web service to persist the state of the application.



Implement Widget data model (5pts)

In **models/Widget.java**, implement a data type that models the concept of user interface widgets described earlier. Group under one Java class all the attributes necessary to represent the various types of widgets. Below is a suggested schema for **Widget.java**. Feel free to add additional fields if needed

Class Variable	Type	Description
name	String	Optional name of the widget
id	Integer	Widget's unique identifier
type	String	Type of the widget, e.g., Heading, List, Paragraph, Image, YouTube, HTML, Link
widgetOrder	Integer	Order with respect to widgets in the same list
text	String	Plain text useful for heading text, paragraph text, link text, etc
src, url, and/or href	String	Absolute or relative URL referring to online resource
size	Integer	Useful to represent size of widget, e.g., heading size
width, height	Integer	Widget's horizontal & vertical size, e.g., Image's or YouTube's width & height
cssClass	String	CSS class implementing some CSS rule and transformations configured in some CSS rule
style	String	CSS transformations applied to the widget
value	String	Some arbitrary initial value interpreted by the widget

Implement service WidgetService.java (20pts)

In **services/WidgetService.java**, implement a service class that manipulates a list of widget instances. Maintain the list of widgets as a Java collection stored in a local class variable. Later assignments will persist the data to a database. A RestfulController will expose the service as a Web service mapping the API of a service to a set of HTTP requests.

Method	Description
Widget createWidget(String tid, Widget widget)	Creates a new Widget instance and add it to the existing collection of widgets for a topic whose ID is tid. Returns new widget with a unique identifier
List<Widget> findWidgetsForTopic(String tid)	Returns collection of all widgets for a topic whose ID is tid
int updateWidget(String wid, Widget widget)	Updates widget whose id is wid encoded as JSON in HTTP body. Returns 1 if successful, 0 otherwise
int deleteWidget(String wid)	Removes widget whose id is wid. Returns 1 if successful, 0 otherwise
List<Widget> findAllWidgets() (optional)	Returns collection of all widgets (optional)
Widget findWidgetById(wid) (optional)	Returns a single widget instance whose id is equal to wid (optional)

Implement RESTful Web service WidgetController.java (20pts)

In **controllers/WidgetController.java**, implement a Web service that provides the following RESTful endpoints mapped to the corresponding handler methods:

HTTP	URL pattern Method	Description
POST	**/api/topics/{tid}/widgets** **Widget createWidget (String tid, Widget widget)** parses Widget JSON object from HTTP body encoded as JSON string. Uses WidgetService to create a new Widget instance and add it to the existing collection of widgets for a topic whose ID is tid. Returns the new widget with a unique identifier	
GET	**/api/topics/{tid}/widgets** **List<Widget> findWidgetsForTopic(String tid)** uses WidgetService to retrieve collection of all widgets and returns a string encoded as a JSON array for a topic whose ID is tid	
PUT	**/api/widgets/{wid}** **int updateWidget(String wid, Widget widget)** parses Widget JSON object from HTTP body encoded as JSON string. Uses WidgetService to find widget instance whose id is equal to wid and update the fields to be the new values in widget parameter. Returns 1 if successful, 0 otherwise.	
DELETE	**/api/widgets/{wid}** **int deleteWidget(String wid)**	

	uses WidgetService to remove widget whose id is wid. Returns 1 if successful, 0 otherwise.
GET	**/api/widgets (optional)**
	List<Widget> findAllWidgets()
	uses WidgetService to retrieve collection of all widgets and returns a string encoded as a JSON array.
GET	**/api/widgets/{wid} (optional)**
	Widget findWidgetById(String wid)
	uses WidgetService to retrieve a single widget instance whose id is equal to wid and returns a string encoded as a JSON object.

Implement RESTful Web service Client WidgetService.js (5pts)

In your React.js project, create a new **services/WidgetService.js** and integrate it to the new WidgetController.java Web service to delete, find, update, and create widgets on the client and store the state on the server. Refactor any React containers, components, and reducers accordingly to use the new implementation of WidgetService.js to integrate with the remote Java server.

Method	Description
createWidget(tid, widget)	**POST**s to **/api/topics/{tid}/widgets** a JSON object encoded as a string representing a new widget instance
findWidgetsForTopic(tid)	**GET**s all widgets as a JSON array from **/api/topics/{tid}/widgets** for a topic whose ID is tid
findAllWidgets() (optional)	**GET**s all widgets as a JSON array from **/api/widgets**
findWidgetById(wid) (optional)	**GET**s a single widget as a JSON object from **/api/widgets/{wid}**
updateWidget(wid, widget)	**PUT**s to **/api/widgets/{wid}** a JSON object encoded as a string representing an existing widget instance with new values for the attributes for a widget whose ID is wid
deleteWidget(wid)	**DELETE**s an existing widget whose ID is wid from **/api/widgets/{wid}**

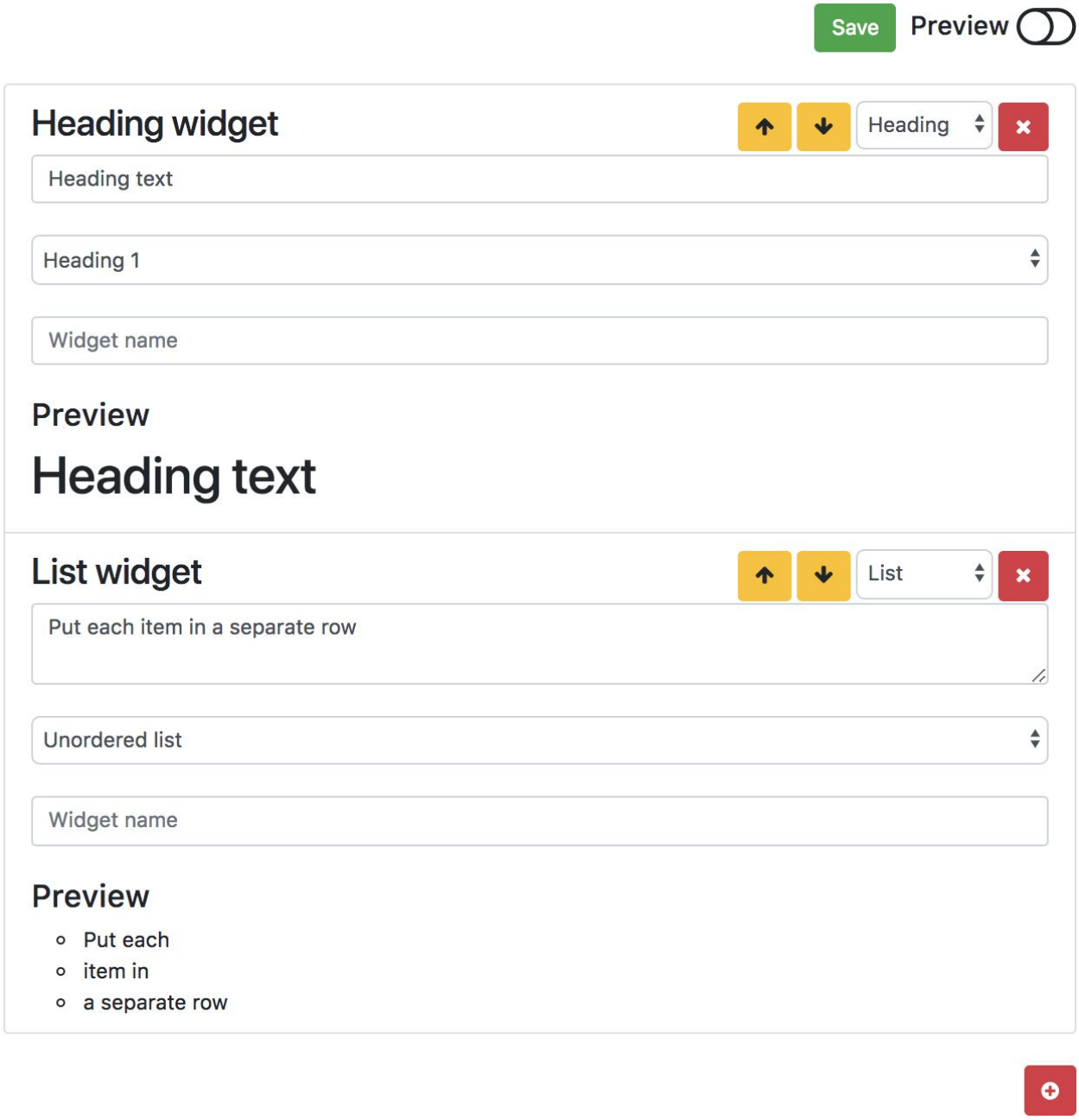
Displaying a list of widgets for a given topic (50pts)

An earlier assignment built a prototype of a list of widgets as hard coded HTML document. This assingment will refactor the implementation to show a different list of widgets for the currently selected topic. Create a component called WidgetList that will show a list of widgets as the user selects a different topic.

Implement WidgetList component (10pts)

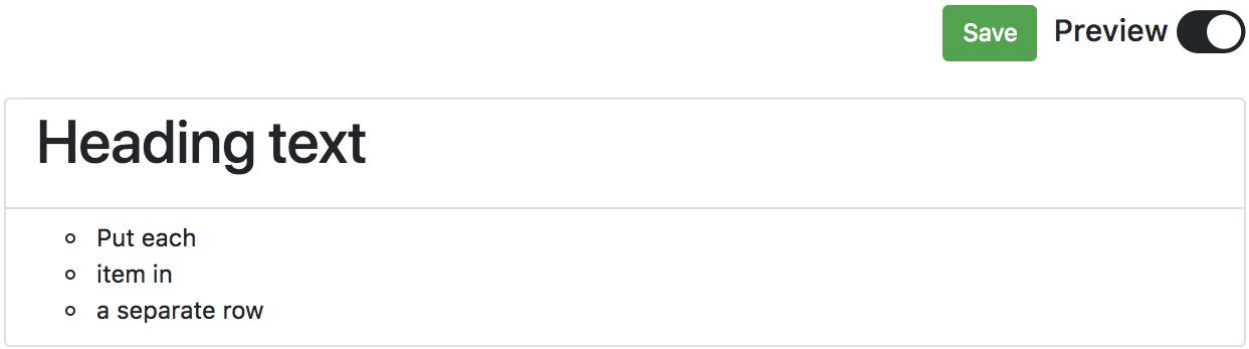
Create React component **WidgetList** to render an array of widgets for the currently selected topic. A different list should display as the user selects a different topic. The component must be connected to a

reducer that provides its state and handles its events. Below is an example of the widget list component rendering the heading and list widgets. There are different types of widgets: Heading, Paragraph, List, Image, Hyperlink, and Video. ***Only implement the Paragraph and Heading*** widgets for this assignment.. Below is an example of the widget list component rendering the heading and list widgets



Clicking the *add widget button* adds a new widget at the bottom of the widget list. The default widget type is Heading of size 1. The user can change the type of the widget using the select *widget type dropdown*. Clicking the *delete widget button* removes that widget from the widget list and the list re-renders without the removed widget. Clicking on the *position up* and *position down buttons* moves the relative position of that widget up or down and the widget list re-renders to give the user feedback that the widget has successfully re-positioned. The *position up button* disappears when the widget is at the top of the widget list and the *position down button* disappears when the widget is at the bottom of the widget list. Clicking the *save widget list button* saves the widget list to the server. The entire list of widgets, with all the configurations, is sent to the server and eventually saved to the database. Refreshing the page before the

widget list is saved will lose all the changes. Clicking the *preview button* changes the widget list view mode to preview in which case only the preview section of each widget is rendered and all editing elements are hidden as shown below



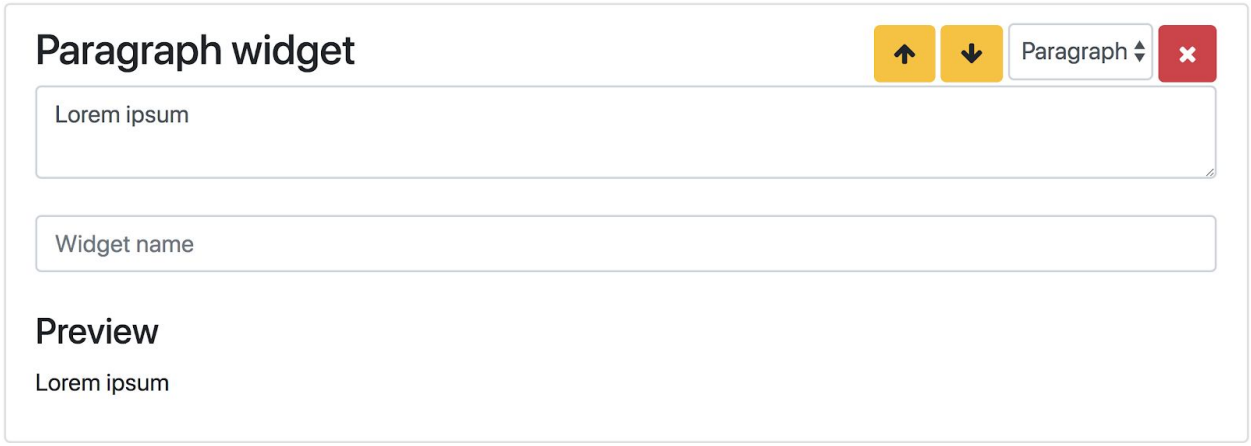
All widgets have a common attribute called *name*. This is an optional attribute that can be useful to refer to the widget from a program, from other widgets, or from anywhere else in the system. Add labels and placeholders for a better user experience (not shown in the wireframes). Changing any of the widget properties updates the widget preview. All widget properties display their current values in the database when rendered from the server.

Each widget has a *widget type dropdown*. The widget's type can be changed at any time by selecting the widget's widget type from the *widget type dropdown*. Changing the widget's type re-renders the widget to match the new widget type with the forms remembering the values the user might have entered.

Alternatively, instead of (or in addition to) the global *save* and *preview buttons*, you may use individual *edit* and *save buttons* for each widget. That is, each widget can have an edit button that toggles it into edit mode showing the form elements and then clicking on the save button would save the widget updates to the server and toggle it back to its preview mode hiding all the form elements only leaving the edit button. In this implementation widgets would be shown in preview mode by default, only showing their preview and an edit button, e.g., a pencil. Clicking on the edit button would hide the edit button and reveal all the other form elements, e.g., type drop down, up, down, delete, save, titles, inputs, text areas, preview, etc. clicking on save would update the widget on the server, hide the form elements, toggle the widget to preview mode, and show the edit button again. Feel free to choose either (or both) implementations.

Paragraph widget (10pts)

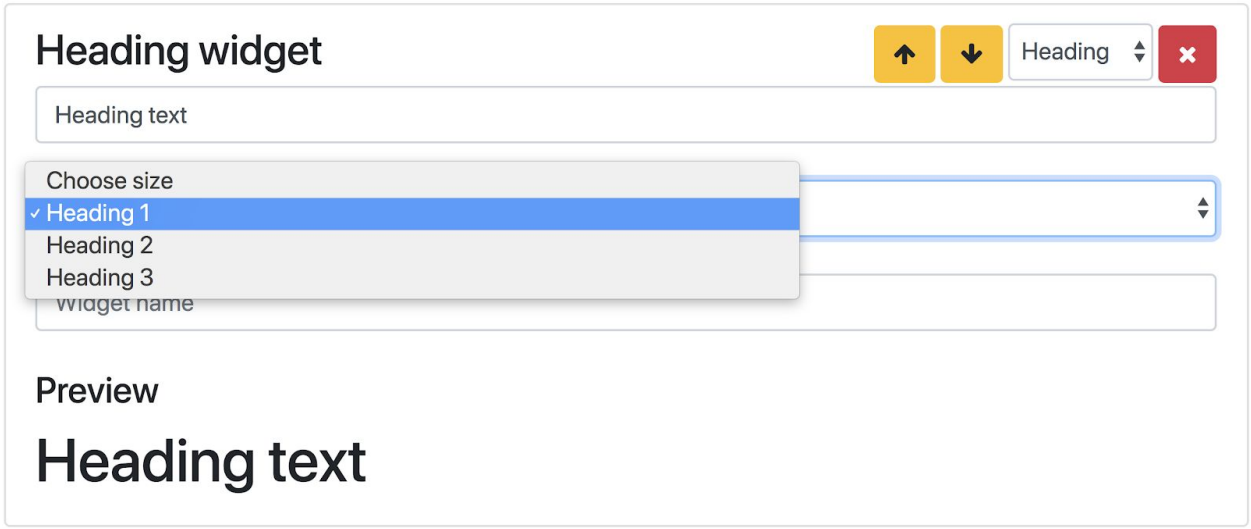
Implement component **ParagraphWidget**. Paragraphs can be added with the paragraph widget where users can enter a text which is rendered as a paragraph as shown below.



The paragraph text property captures the text in the paragraph. Try to use a vertically resizable textbox to enter the paragraph text. The placeholder should be *Paragraph text (mentioned as Lorem ipsum in the above picture)*.

Heading widget (10pts)

Implement component **HeadingWidget**. Headings can be added with the heading widget of which the user can select from six different sizes (only 3 shown below).



The heading size widget property configures the heading's size. The default size is Heading 1. The heading text widget property configures the widget's text. The default text is an empty string and the placeholder should be *Heading text*. This assignment updates two repositories and two Heroku remote servers. Submit a link to all relevant Git repositories and heroku remote servers.

Implement a WidgetReducer (10pts)

In **src/reducers/widgetReducer.js** create a Redux reducer that implements the following action type:

Action Types	Description
CREATE_WIDGET	creates a new widget instance for the topic whose ID is topicId
DELETE_WIDGET	removes an existing widget

UPDATE_WIDGET	updates an existing widget
FIND_ALL_WIDGETS_FOR_TOPIC	retrieves all widgets for a particular topic
FIND_ALL_WIDGETS (optional)	retrieves all widgets
FIND_WIDGET (optional)	retrieves a particular widget

Add additional actions as needed, e.g., WIDGET_POSITION_UP, WIDGET_POSITION_DOWN, PREVIEW_ON, PREVIEW_OFF, SAVE_ALL, etc. Each widget can hold local internal state and handle internal events if necessary.

Conventions, naming, file name and directory names (10pts)

Client side conventions

- ######Component
    - Stateless React JS components should end with Component
    - Implement stateless React JS components as ES6 arrow functions
    - Stateful React JS components which use the state for internal layout and behavior should end with Component
    - All components should be in a folder called components
- ######Container
    - Stateful React JS components which provide state and event handlers for childd elements should end with Container
    - All container components should be in a folder called containers
- ######Service
    - All communication between client and server should be implemented in separate service files
    - The name of service files should end with Service
    - All services should be in a folder called services

Server side conventions

- ######Controller
    - All controllers should be in a folder called controllers
    - Defines URLs to expose a data model as web services
    - Defines POST end point to create data
    - Defines GET end point to retrieve data
    - Defines PUT end point to update data
    - Defines DELETE end point to remove data
    - URLs contain plural nouns of data being manipulated
    - Uses services to manipulate data model
    - Singleton
- ######Service
    - All services should be in a folder called services
    - Declares CRUD operations to manipulate data model
    - Agnostic to protocol
    - Singleton

Deliverables

As a deliverable, check in all your work into a github source control repository. Deploy your project to a remote public server on Heroku or AWS. Submit the URL of all repositories and remote servers in blackboard. Tag the commit you want graded as assignment5. TAs will clone your repository and grade the tagged commit.