| Topic	METEOR SCREEN 1
Class Description	**Students learn to generate an API key and calculate threat scores for the meteors after fetching the data from the API.**
Class	**C79**
Class time	**45 mins**
Goal	● Generate the API key. ● Get the data using the API key. ● Calculate the threat score for the meteors
Resources Required	● Teacher Resources: 　○ Visual Code Studio Editor 　○ laptop with internet connectivity 　○ earphones with mic 　○ notebook and pen ● Student Resources: 　○ Visual Code Studio Editor 　○ laptop with internet connectivity 　○ earphones with mic 　○ notebook and pen

Class structure	Warm Up Teacher & Student Collaborative Activity Wrap up	5 mins 30 mins 5 min
Credits	**● Open source API for getting updates on meteors offered by Nasa's open repository APIs.**	

WARM UP SESSION - 5 mins

Teacher starts slideshow **from slides 1 to 13**

Refer to speaker notes and follow the instructions on each slide.	
Teacher Action	**Student Action**
Hey <student name>. How are you? It's great to see you! Are you excited to learn something new today? **Run the presentation from slide 1 to slide 3.** **Following are the warm up session deliverables:** ● Connecting students to the previous class.	**ESR**: Hi, thanks, yes I am excited about it! Click on the slide show tab and present the slides.
QnA Session	
Question	**Answer**
Choose the correct block of code that can be used to import axios from the axios library. A. `import axios from 'axios';` B. `import {axios} from 'axios';` C. `import axios from axios;` D. `import "axios" from axios;`	**A**
Choose the correct block of code to render the Image component.	**A**
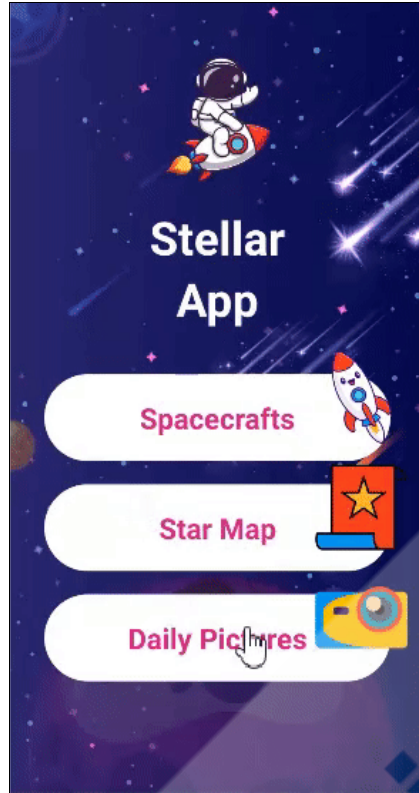
A.
```
<Image source={{ "uri": url }}
  style={{ width: "100%", height: 300,
borderRadius: 20, margin: 3 }}>
  </Image>
```

B.
```
<Image source={ "uri": url }
  style={{ width: "100%", height: 300,
borderRadius: 20, margin: 3 }}>
  </Image>
```

C.
```
<Image source={{ "uri"= url }}
  style={{ width: "100%", height: 300,
borderRadius: 20, margin: 3 }}>
  </Image>
```
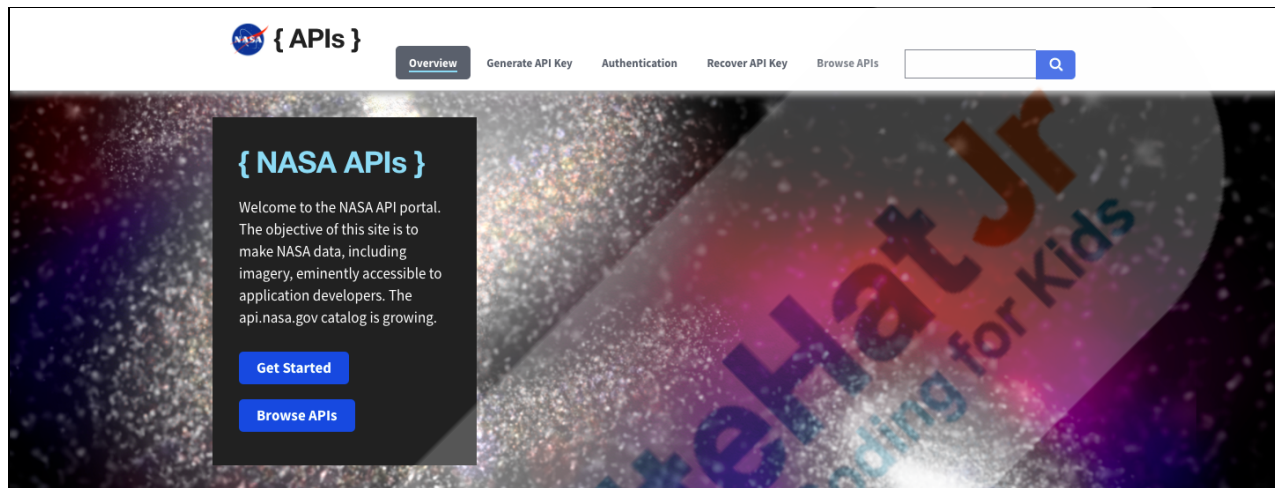
D.
```
<Image require={{ "uri": url }}
  style={{ width: "100%", height: 300,
borderRadius: 20, margin: 3 }}>
  </Image>
```

Continue the warm up session	
Teacher Action	**Student Action**
Run the presentation from slide 4 to slide 13 to set the problem statement. **Following are the warm up session deliverables:** ● Talk about the different sizes and threats of the meteor	Narrate the story by using hand gestures and voice modulation methods to bring in more interest in students.

Teacher ends slideshow

TEACHER & STUDENT COLLABORATIVE ACTIVITY - 30 mins

Teacher Initiates Screen Share

ACTIVITY
● **Get the API key by signing up on the official website.** ● **Write a function to get data from an API.**

Teacher Action	**Student Action**
<The teacher opens the code from the previous class.> *<Teacher should take note that this is a collaborative class. The student is expected to code with Teacher's guidance.>*	
Alright, so before we even begin to write the tiniest bit of code, let's first see where we are going to find the data of meteors. For that, please refer to the *Student Activity 1.* *<Teacher opens the Teacher Activity 1.>*	

Here, the first thing we'll do is, we'll get started to sign up for the API. This way, NASA will allow us to use their API.	

When we click on "Get started button", it takes us to the following form -	

Let's fill this form quickly so that we can generate our respective API keys. *<Teacher fills the Form.>*	*<Student fills the form.>*
On filling the form and submitting it, we can see that it takes you to a screen where it displays your API Key.	

```
3MT0uxLui64D9olxQOlsxfvFK1Lw1b1HJJZ5LzA5
```

We will have to save this API key somewhere, as we'll be using this later. Now, in a separate tab, let's open Nasa's website again. *<Teacher opens the link in Teacher Activity 1 in a different tab.>*	*<Student opens the link in Student Activity 1 in a different tab.>*
If you scroll down a bit on Nasa's website, you'll find the following section -	

Browse APIs

	Search

APOD: Astronomy Picture of the Day   +

Asteroids NeoWs: Near Earth Object Web Service   +

DONKI: Space Weather Database Of Notifications, Knowledge, Information   +

Earth: Unlock the significant public investment in earth observation data   +

EONET: The Earth Observatory Natural Event Tracker   +

EPIC: Earth Polychromatic Imaging Camera   +

Exoplanet: Programmatic access to NASA's Exoplanet Archive database   +

GeneLab: Programmatic interface for GeneLab's public data repository website   +

Insight: Mars Weather Service API   +

Mars Rover Photos: Image data gathered by NASA's Curiosity, Opportunity, and Spirit rovers on Mars   +

We will be using the second one, which states "**Asteroids NeoWs**". Let's expand this one -

Asteroids - NeoWs

NeoWs (Near Earth Object Web Service) is a RESTful web service for near earth Asteroid information. With NeoWs a user can: search for Asteroids based on their closest approach date to Earth, lookup a specific Asteroid with its NASA JPL small body id, as well as browse the overall data-set.

Data-set: All the data is from the NASA JPL Asteroid team (http://neo.jpl.nasa.gov/).

This API is maintained by SpaceRocks Team: David Greenfield, Arezu Sarvestani, Jason English and Peter Baunach.

Neo - Feed

Retrieve a list of Asteroids based on their closest approach date to Earth. `GET`
`https://api.nasa.gov/neo/rest/v1/feed?`
`start_date=START_DATE&end_date=END_DATE&api_key=API_KEY`

Query Parameters

Parameter	Type	Default	Description
start_date	YYYY-MM-DD	none	Starting date for asteroid search
end_date	YYYY-MM-DD	7 days after start_date	Ending date for asteroid search
api_key	string	DEMO_KEY	api.nasa.gov key for expanded usage

As we can see, this is an API Nasa provides where we can specify the **start_date** and the **end_date**. We also need our **api_key**, the one that we just generated.

If we look at the default values in the table mentioned below, we can see that it takes the default values as -

1. start_date - the current date
2. end_date - 7 days after the start date

This means that if we don't provide the start and the end date, we will automatically receive meteor data for the next 7 days.

Don't forget that we still have to provide the API keys that we just generated. Now, let's take a look at the data that we are receiving. For this, refer to the link in *Student Activity 2* and replace the "DEMO_KEY" with your API key. *<Teacher refers to [Teacher Activity 2](#) and replaces the DEMO_KEY with their API key.>*	*<Student refers to [Student Activity 2](#) and replaces the DEMO_KEY with their API key.>*

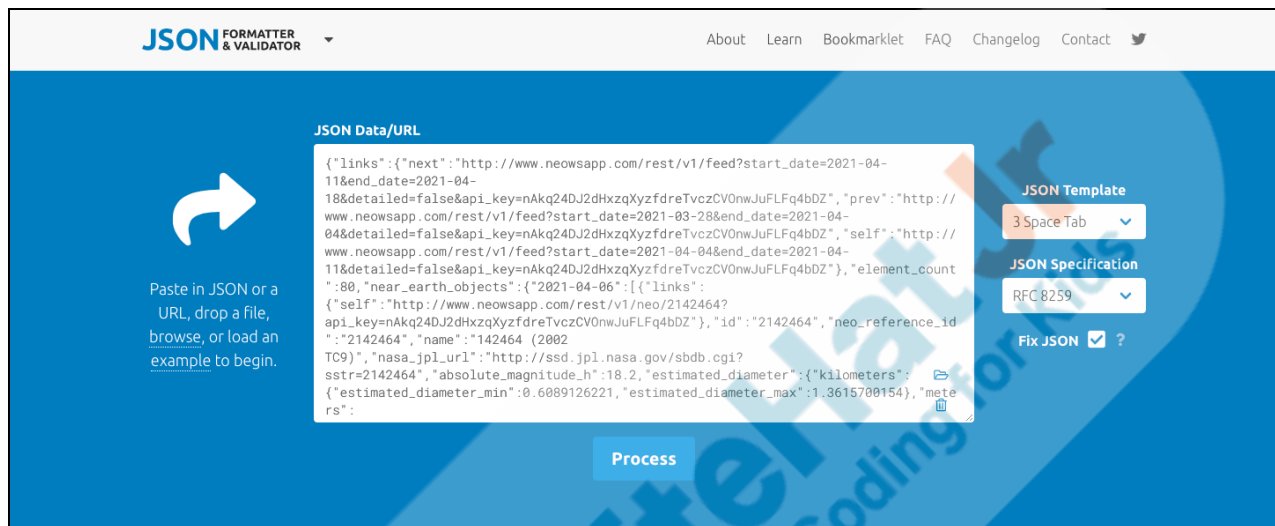|	
That's a lot of data! Like this, I don't think we'll be able to figure out what all it is sending us, but we can find a smarter way. Let's use JSON Prettify Tools, which is available online. We can simply copy and paste this data in there and it will display the data in an organized way.	

© 2021 - WhiteHat Education Technology Private Limited.

Please refer to *Student Activity 3.* Copy this data and paste it there, then click on "Process".

<Teacher refers to Teacher Activity 3 and does the same.>

<Student refers to Student Activity 3 and does the same.>

That looks way more organized than before! Let's see this data and try to figure out how it's structured.

The first thing that you'll notice is that this data has a key called **near_earth_objects** which contains the dates of the next 7 days as keys.

Each of these dates has an array as a value, which contains the data for meteors on these dates.

In the data that we receive for each of the meteors, we can see that it has a key called **estimated_diameter** which contains the estimated minimum and maximum diameter in different units - kilometer, meter, miles, feet.

```
"estimated_diameter":{ ⊟
    "kilometers":{ ⊟
        "estimated_diameter_min":0.6089126221,
        "estimated_diameter_max":1.3615700154
    },
    "meters":{ ⊟
        "estimated_diameter_min":608.9126221057,
        "estimated_diameter_max":1361.5700153859
    },
    "miles":{ ⊟
        "estimated_diameter_min":0.3783606449,
        "estimated_diameter_max":0.846040122
    },
    "feet":{ ⊟
        "estimated_diameter_min":1997.7448871092,
        "estimated_diameter_max":4467.0933692788
    }
},
```

We also have a key called **close_approach_data** that gives us some great data like the date at which it will be closest to Earth, it's relative velocity and also, the distance by which it's missing Earth.

```
"close_approach_data":[ ⊟
    { ⊟
        "close_approach_date":"2021-04-06",
        "close_approach_date_full":"2021-Apr-06 19:55",
        "epoch_date_close_approach":1617738900000,
        "relative_velocity":{ ⊟
            "kilometers_per_second":"8.6691248134",
            "kilometers_per_hour":"31208.8493282193",
            "miles_per_hour":"19391.9652950816"
        },
        "miss_distance":{ ⊟
            "astronomical":"0.1333993391",
            "lunar":"51.8923429099",
            "kilometers":"19956256.988767717",
            "miles":"12400243.0859078146"
        },
        "orbiting_body":"Earth"
    }
],
```

Now, that's awesome!

We now have enough data to build our screen, so let's start coding!

The first thing that we want to do is to import axios and then make a get request to the already existing meteor screen.

Student-led Activity (with Teacher's help)

We know the URL we want to make the request to, to get data for meteors and you've already made a GET request in the previous class, so I guess you can help me out with this?

The teacher helps the student in writing the code to make a get request to fetch all the meteor data.

```
import React, { Component } from 'react';
import { Text, View } from 'react-native';

import axios from "axios";

export default class MeteorScreen extends Component {

    getMeteors = () => {
        axios
            .get("https://api.nasa.gov/neo/rest/v1/feed?api_key=nAkq24DJ2dHxzqXyzfdreTvczCVOnwJ
            .then(response => {
                this.setState({ meteors: response.data.near_earth_objects })
            })
            .catch(error => {
                Alert.alert(error.message)
            })
    }

    render() {
        return (
            <View
                style={{
                    flex: 1,
                    justifyContent: "center",
                    alignItems: "center"
                }}>
                <Text>Meteor Screen!</Text>
            </View>
        )
    }
}
```

If you note here, we are saving just the **near_earth_objects** value that contained all our data in our state.

Since we want to save our data as a state in variable **meteors**, let's create a constructor.

Student-led Activity (with Teacher's help)

Also, let's not forget to add our **getMeteors()** function in a **componentDidMount()**, since that's the first thing we want to happen as soon as someone visits this screen!	*The teacher helps the student in writing the code to create the constructor and the componentDidMount() function*

```
constructor(props) {
    super(props);
    this.state = {
        meteors: {},
    };
}


componentDidMount() {
    this.getMeteors()
}
```

Awesome! Now, we are ready with our data.

Now since we are making an API call, it could be that the data is not fetched yet and our app is trying to render the UI. In that case, our app will break.

For that, we added a condition in the last class where we were checking if we had received the data or not.

Student-led Activity (with Teacher's help)

We'll display "Loading"; if not, we will continue with the data. Let's do that here as well -	*The teacher helps the student in writing the if-else statements in the render function.*

```
render() {
    if (Object.keys(this.state.meteors).length === 0) {
        return (
            <View
                style={{
                    flex: 1,
                    justifyContent: "center",
                    alignItems: "center"
                }}>
                <Text>Loading</Text>
            </View>
        )
    } else {
        return (
            <View
                style={{
                    flex: 1,
                    justifyContent: "center",
                    alignItems: "center"
                }}>
                <Text>Meteor Screen!</Text>
            </View>
        )
    }
}
```

Here, we are using the **Object.keys()** method which checks if there are any keys in our meteors state or not. If it finds 0 keys, it will display loading otherwise it will display our meteor screen text.	
Now comes the most important part! What do we want to display on our meteors screen? To give you an idea, have a look at this UI -	**ESR:** We want to display the information of the meteor ● when it will be closest to Earth; ● what is its minimum diameter in KM; ● what is its maximum diameter in KM; ● its velocity; and ● how much distance will it miss the Earth by in KM.

(1997 GL3)

Closest to Earth - 2021-Apr-09 14:18
Minimum Diameter (KM) - 0.4023045798
Maximum Diameter (KM) - 0.8995803882
Velocity (KM/H) - 95372.0028547211
Missing Earth by (KM) - 10389252.859564994

	ESR:
Looks good doesn't it? That's exactly what we are going to build. Now our data was structured in a very weird way, and we had a lot of data. We will first need to organize our data. Once it is organized, we will have to find a way to reduce the number of meteors we are displaying because otherwise displaying so many meteors doesn't make sense, right?	Yes!
What if we come up with a way where we can calculate a **threat score** for a meteor, which will tell how threatful it is to Earth.	**ESR:** Yes!

Based on this **threat score,** we can take the top 5 and display just those.	
Let's think about the data that we have. We know that we have the keys as dates and their values as a list of objects containing meteor data. Our end result should be a master array that contains all the objects with meteor data. Can you think of a way of how we can do this?	**ESR:** Varied.
For this, we can first iterate over all the keys in the object, which are the dates of the next 7 days. We can take their values, which is a list of objects containing meteor data and save these lists inside another array. **Student-led Activity (with Teacher's help)** Once we are done, we will have an array of arrays! Now all we need to do is to pick all the objects inside all these arrays and put them in an array! Let's do that -	*Teacher helps the student in writing the code for the same and explains the code.*

```
    } else {
        let meteor_arr = Object.keys(this.state.meteors).map(meteor_date => {
            return this.state.meteors[meteor_date]
        })
        let meteors = [].concat.apply([], meteor_arr);

        return (
            <View
                style={{
                    flex: 1,
                    justifyContent: "center",
                    alignItems: "center"
                }}>
                <Text>Meteor Screen!</Text>
            </View>
        )
    }
```

Remember that we are doing this inside the else statement in our render() function, so it gets executed every time our app renders the meteor screen.

Here, we are first creating a **meteor_arr** which contains all the array of objects on a particular **meteor_date** with the **map()** function by iterating over all the keys of the object (which are the dates of the next 7 days).

Next, we are concatenating all the arrays inside our **meteor_arr** into another master array with the **concat.apply()** function, and we are storing it inside another variable called **meteors**.

Now this variable, **meteors**, contains our array of objects for all the meteors.

Next, we need to calculate the threat score. Let's understand how we can do it!

Can you think of the data we should consider while calculating the threat score?	**ESR:** Varied!
We can take the diameter and the distance by which a meteor will miss the Earth. Why? Well, we know that the diameter of the array can be taken in kilometers. We also know that the distance by which it will miss Earth can also be taken in kilometers since both these values are in kilometers. Since both the values are in kilometers, if we divide the diameter of the meteor with the distance by which it misses Earth, we can cancel out the SI unit kilometer of the diameter with the SI unit kilometer of the distance by which it's missing Earth. That way, it will leave us with a scalar quantity. SI unit is just the unit of measure for a particular thing. For example, time can be measured in seconds, distance in meters, speed in meters per second, etc. Now, to sum up this discussion, we can say that -	
threat_score = diameter/distance by which the meteor misses Earth	
Now, since we have a minimum and maximum diameter of the meteor in our data, we can take the average of it by adding the minimum and maximum diameter and dividing it by 2.	

In case of an international student, use distance.miles in calculating the Threat score. Our code now looks something like the following in the else condition-	*Teacher helps student in writing the code to calculate the threat score for all the meteors*

```
let meteor_arr = Object.keys(this.state.meteors).map(meteor_date => {
    return this.state.meteors[meteor_date]
})
let meteors = [].concat.apply([], meteor_arr);

meteors.forEach(function (element) {
    let diameter = (element.estimated_diameter.kilometers.estimated_diameter_min + element.estimated_diameter.kilometers.estimated_diameter_max) / 2
    let threatScore = (diameter / element.close_approach_data[0].miss_distance.kilometers) * 1000000000
    element.threat_score = threatScore;
});
```

Additional code added for threat score logic (can be copy pasted and then understood) -

```
meteors.forEach(function (element) {
            let diameter =
(element.estimated_diameter.kilometers.estimated_diameter_min +
element.estimated_diameter.kilometers.estimated_diameter_max) / 2
            let threatScore = (diameter /
element.close_approach_data[0].miss_distance.kilometers) * 1000000000
            element.threat_score = threatScore;
        });
```

Here, we are iterating over the objects of meteors with the **map()** function and we are first calculating the average diameter. We are then calculating the threat score with the formula we discussed above. You will notice that we are multiplying `1000000000` to our threat score. This is because most of the values are in billions to be precise and we did a pre-analysis of the data for you. To get most of the values above 0, we are multiplying that huge number.	

Once we calculate the threat score, we are adding their respective threat score in their objects.	
Phew! Well, that was a lot of code but we are now set to use the meteor data in the arrays to generate the screen UI as we noticed above. In the next class, we will finish building these screens.	

WRAP UP SESSION - 5 Mins

Teacher starts slideshow **from slide 13 to slide 23**

Activity details	Solution/Guidelines
Run the presentation from slide 14 to slide 23 **Following are the warm up session deliverables:** ● **Explain the facts and trivia** ● **Next class challenge** ● **Project for the day** ● **Additional Activity**	Guide the student to develop the project and share with us.
Let's quickly wrap up today's class. What did we learn?	**ESR:** We created the meteor arrays with threat scores! We generated an API key and used the API to get the data for meteors and then performed logic on it that will help us create the screen in the next class!

Quiz time - Click on in-class quiz

Question	Answer
Which of the following components is used to avoid the status bar? A. SafeAreaView B. ImageBackground C. StatusBar D. Maps	A
What does the JSON Prettify Tools do? A. It will convert the data into JSON type. B. It will display the data in an organized way. C. It will display the data in JSON after converting. D. It will delete the unnecessary data in JSON.	B

In the following code snippet, what does the if condition do?
 	
``
render() {
    if (Object.keys(this.state.meteors).length === 0) {
        return (
            <View
                style={{
                    flex: 1,
                    justifyContent: "center",
                    alignItems: "center"
                }}>
                <Text>Loading</Text>
            </View>
``
 	
A. it checks if there are any keys in our meteor's state or not
B. it checks if there are 0 keys in our meteors state
C. it checks the length of the keys in the meteor state
D. all of the above
	D

End the quiz panel	

FEEDBACK	
● **Appreciate the student for their efforts in the class.** ● **Ask the student to make notes for the reflection journal along with the code they wrote in today's class.**	

Teacher Action	Student Action
Did you enjoy today's class?	**ESR:** Varied.
Amazing work today! You get a "hats-off".	*Make sure you have given at least 2 Hats Off during the class for:* Creatively Solved Activities +10 Great Question +10 Strong Concentration +10
In the next class, we will work on completing our meteor screen and with it, our ISS Tracker app will get completed.	
Alright. See you in the next class.	
Project Overview	** This Project will take only 30 mins to complete. Motivate students to try and finish it immediately after the class.* **Stellar Stage 4** **Goal of the Project:**	

In Class 79, we have designed the meteors screen to show all the meteors that are going close to the Earth.
In this project, you will create a similar "Daily Pic" screen for the Stellar App.

*This is a continuation project of Projects-76, 77 & 78. Make sure to complete that one before attempting this one.

Story:

Jeff is impressed with the constellation's screen output; he wants to use a similar feature to highlight daily pictures received by NASA on space happenings. He would like you to create a separate screen for that.

I am very excited to see your project solution and I know you will do really well. Bye Bye!

Teacher ends slideshow

Teacher Clicks ✖ End Class

ADDITIONAL ACTIVITY

Activity	Activity Name	Links
Teacher Activity 1	Nasa's website	https://api.nasa.gov/
Teacher Activity 2	API URL	https://api.nasa.gov/neo/rest/v1/feed?api_key=DEMO_KEY
Teacher Activity 3	JSON Prettifier	https://jsonformatter.curiousconcept.com/
Teacher Activity 4	Reference code	https://github.com/pro-whitehatjr/C79_ISSTracker_TeacherReferenceCode
Teacher Activity 5	Teacher Aid	https://drive.google.com/file/d/1WA1BQff4dmgv5BInU3f_imk4vlpvAyMa/view?usp=sharing
Student Activity 1	Nasa's website	https://api.nasa.gov/
Student Activity 2	API URL	https://api.nasa.gov/neo/rest/v1/feed?api_key=DEMO_KEY
Student Activity 3	JSON Prettifier	https://jsonformatter.curiousconcept.com/
Teacher Reference visual aid link	Visual aid link	https://curriculum.whitehatjr.com/Visual+Project+Asset/PRO_VD/BJFC-PRO-V3-C79-withcues.html
Teacher Reference In-class quiz	In-class quiz	https://s3-whjr-curriculum-uploads.whjr.online/9b105400-2d3d-4931-9ad5-575b70e5af15.pdf
Project Solution	Stellar Stage-4	https://github.com/pro-whitehatjr/Stellar-Stage-4