UNIT I

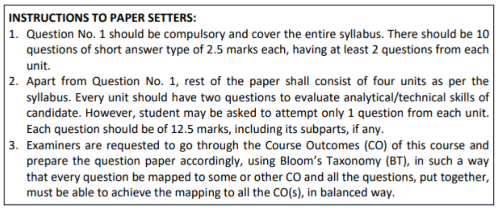
Full Stack Development

Learning Resources

- Books
  - A. Banks and E. Porcello, "Learning React: Functional Web Development with React and Redux", O'Reilly, 1st Edition, 2017.
- Web Links (Strictly Referred):
  - https://reactjs.org/
  - https://nodejs.org/
  - https://expressjs.com/
  - https://developer.mozilla.org
  - https://react-redux.js.org/

Learning Objectives

- The core concepts of both the frontend and backend programming.
- The latest web development technologies.
- Maintaining data using NoSQL data bases.
- Complete web application development process


INSTRUCTIONS TO PAPER SETTERS:

1. Question No. 1 should be compulsory and cover the entire syllabus. There should be 10 questions of short answer type of 2.5 marks each, having at least 2 questions from each unit.
2. Apart from Question No. 1, rest of the paper shall consist of four units as per the syllabus. Every unit should have two questions to evaluate analytical/technical skills of candidate. However, student may be asked to attempt only 1 question from each unit. Each question should be of 12.5 marks, including its subparts, if any.
3. Examiners are requested to go through the Course Outcomes (CO) of this course and prepare the question paper accordingly, using Bloom's Taxonomy (BT), in such a way that every question be mapped to some or other CO and all the questions, put together, must be able to achieve the mapping to all the CO(s), in balanced way.

Course Outcome

- CO1: Relate the basics of Javascript (JS) and ReactJS
- CO2: Apply the concepts of props and State Management in React JS
- CO3: Examine Redux and Router with React JS
- CO4: Appraise Node JS environment and modular development
- CO5: Develop full stack applications using MongoDB

Overview

UNIT-1

- Introduction to React
  - Obstacles and Roadblocks
    - React Library, React Developer tools
  - Introduction to ES6
    - Declaring variables in ES6, Arrow Functions, ES6 Objects and Arrays, ES6 modules
  - Introduction to AJAX
- Pure React
  - Page setup, virtual DOM
  - React Element, React DOM, Constructing Elements with Data, React Components, DOM Rendering, First React Application using Create React App, React with JSX, React Element as JSX
- Props, State and Component Tree
  - Property Validation, Validating Props with createClass, Default Props, ES6 Classes and stateless functional components, React state management, State within the component tree, state vs. props, Forms in React

Overview (cont..)

UNIT-2

- Enhancing Components
  - Component Lifecycle, JavaScript library integration
  - Higher-Order Components, Managing state outside the react
  - Introduction to Flux
- Redux and Router
  - State, Actions, Reducers, The Store
  - Middleware
  - React Redux
  - React Router, Incorporating the router, Nesting Router, Router parameters
- JSON
  - Objects
  - Schema
- REST API
  - WRML, REST API Design
  - Identifier Design with URIs, Interaction Design with HTTP, Representation Design, Caching, Security

Overview (cont..)

UNIT-3

•Introduction to Angular
  •Angular architecture; introduction to components, component interaction and styles; templates, interpolation and directives; forms, user input, form validations; data binding and pipes; retrieving data using HTTP; Angular modules

•Node.js
  • Introduction, Features, Node.js Process Model
  •Environment Setup: Local Environment Setup, The Node.js Runtime, Installation of Node.js
  •Node.js Modules: Functions, Buffer, Module, Modules Types
  •Node Package Manager: Installing Modules using NPM, Global vs Local Installation, Attributes of Package.js on, Updating packages, Mobile-first paradigm, Using twitter bootstrap on the notes application, Flexbox and CSS Grids

•File System: Synchronous vs Asynchronous, File operations
•Web Module: Creating Web Server, Web Application Architecture, Sending Requests, Handling http requests
•Express Framework: Overview, Installing Express, Request / Response Method, Cookies Management

Overview (cont..)

UNIT-4

•MongoDB:
  •Introduction to NoSQL
  •Understanding MongoDB datatypes
  •Building MongoDB Environment (premise and cloud based)
  •Administering Databases and User accounts
  •Configuring Access Control, Managing Collections
  •connecting to MongoDB from Node.js
  •Accessing and Manipulating Databases and Collections
  •Manipulating MongoDB documents from Node.js
  •Understanding Query objects,
  •sorting and limiting result sets

Component Lifecycles

- Mounting Lifecycle
  - invoked when a component is mounted or unmounted
  - these methods allow you to initially set up state, make API calls, start and stop timers, manipulate the rendered DOM, initialize third-party libraries

ES6 class	React.createClass()
	getDefaultProps()
constructor(props)	getInitialState()
componentWillMount()	componentWillMount()
render()	render()
componentDidMount()	componentDidMount()
componentWillUnmount()	componentWillUnmount()

Component Lifecycles

- Mounting Lifecycle
  - constructor(): If you don’t initialize state and you don’t bind methods, you don’t need to implement a constructor for your React component.
  - componentWillMount(): incokes before the render().
  - getDerivedStateFromProps(): invoked right before calling the render method
    - return an object to update the state, or null to update nothing
  - render()
  - componentDidMount(): invoked immediately after a component is mounted

Component Lifecycles

- Update Component Lifecycle
  - componentWillReceiveProps(nextProps): invoked if new properties have been passed to the component
  - getDerivedStateFromProps():invoked right before calling the render method
  - shouldComponentUpdate(): React know if a component’s output is not affected by the current change in state or props
  - render()
  - getSnapshotBeforeUpdate(): invoked right before the most recently rendered output is committed to e.g. the DOM
    - It enables your component to capture some information from the DOM (e.g. scroll position) before it is potentially changed
  - componentWillUpdate(nextProps, nextState): Invoked just before the component updates
  - componentDidUpdate(prevProps, prevState): invoked immediately after updating occurs
    - This method is not called for the initial render

Component Lifecycles

- Unmounting Lifecycle
  - componentWillUnmount(): is invoked immediately before a component is unmounted and destroyed

JavaScript Library Integration

- React
  - simply a library for creating views
  - Fetch
  - D3 Timeline

JavaScript Library Integration

- Making Requests with Fetch
  - Fetch is a polyfill created by the WHATWG group
    - allows us to easily make API calls using promises
  - npm install isomorphic-fetch –save
  - component lifecycle functions provide us a place to integrate JavaScript
  - they are where we will make an API call
  - handle latency, the delay that the user experiences while waiting for a response

```
componentDidMount() {
    this.setState({loading: true})
    fetch('https://restcountries.eu/rest/v1/all')
        .then(response => response.json())
        .then(json => json.map(country => country.name))
        .then(countryNames =>
            this.setState({countryNames, loading: false})
        )
}
```

JavaScript Library Integration

- Incorporating a D3 Timeline
  - Data Driven Documents (D3) is a JavaScript framework that can be used to construct data visualizations for the browser
  - D3 is functional
  - `npm install d3 --save`
  - D3 takes data, typically arrays of objects, and develops visualizations based upon that data

```
const historicDatesForSkiing = [
    {
        year: 1879,
        event: "Ski Manufacturing Begins"
    },
    {
        year: 1882,
        event: "US Ski Club Founded"
    },
```

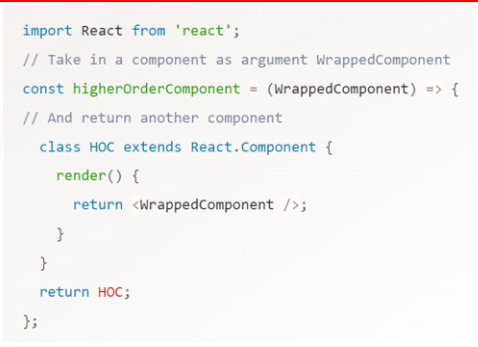
Learning React: Functional Web Development with React and Redux Page No. 160

Higher-Order Components

- a function that takes a React component as an argument and returns another React component
- best way to reuse functionality across React components
- The parent component can hold state or contain functionality that can be passed down to the composed component as properties

```
render(
    <PeopleList />,
    document.getElementById('react-container')
)
```

Higher-Order Components



```
import React from 'react';
// Take in a component as argument WrappedComponent
const higherOrderComponent = (WrappedComponent) => {
// And return another component
  class HOC extends React.Component {
    render() {
      return <WrappedComponent />;
    }
  }
  return HOC;
};
```

Managing State Outside of React

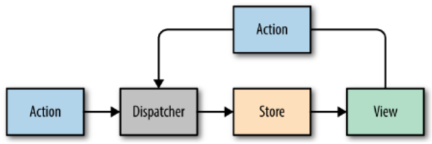
- benefits of managing state outside of React is:
  - reduce the need for many, if any, class components
  - state full component
  - you can isolate class functionality to HOCs
  - Stateless functional components are easier to understand and easier to test
  - create your own system for managing state
  - manage state using global variables or localStorage and plain JavaScript
  - Managing state outside of React simply means not using React state or setState in your applications

Learning React: Functional Web Development with React and Redux Page No. 173

Flux

- Flux is a design pattern developed at Facebook that was designed to keep data flowing in one direction
- Before Flux, MVC design pattern was used
- a stateless functional component is a pure function
- const Countdown = ({count}) => <h1>{count}</h1>
- application state data is managed outside of React components in stores
- Stores hold and change the data, and are the only thing that can update a view in Flux
- If a user were to interact with a web page an action would be created to represent the user's request
- Actions are dispatched using a central control component called the dispatcher

Flux



Views

- a React stateless component
- Flux will manage application state
- a lifecycle function will not need class component

```
const Countdown = ({count, tick, reset}) => {

  if (count) {
    setTimeout(() => tick(), 1000)
  }

  return (count) ?
      <h1>{count}</h1> :
      <div onClick={() => reset(10)}>
          <span>CELEBRATE!!!</span>
          <span>(click to start over)</span>
      </div>

}
```

Actions and Action Creators

- Actions provide the instructions and data that the store will use to modify the state.
- Action creators are functions that can be used to abstract away the nitty-gritty details required to build an action.
- Actions themselves are objects that at minimum contain a type field.
- The action type is typically an uppercase string that describes the action.

```
const countdownActions = dispatcher =>
    ({
        tick(currentCount) {
            dispatcher.handleAction({ type: 'TICK' })
        },
        reset(count) {
            dispatcher.handleAction({
                type: 'RESET',
                count
            })
        }
    })
```

Dispatcher

- dispatcher takes the action, packages it with some information about where the action was generated and sends it on to the appropriate store or stores that will handle the action

```
import Dispatcher from 'flux'

class CountdownDispatcher extends Dispatcher {

    handleAction(action) {
        console.log('dispatching action:', action)
        this.dispatch({
            source: 'VIEW_ACTION',
            action
        })
    }

}
```

Stores

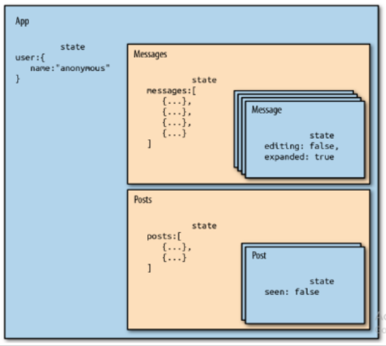
- Stores are objects that hold the application's logic and state data
- Stores are similar to models in the MVC pattern
  - stores are not restricted to managing data in a single object
- Current state data can be obtained from a store via properties
- A store will handle actions by type and change their data accordingly

```
import { EventEmitter } from 'events'

class CountdownStore extends EventEmitter {

    constructor(count=5, dispatcher) {
        super()
        this._count = count
        this.dispatcherIndex = dispatcher.register(
            this.dispatch.bind(this)
        )
    }
```

State

- application state should be stored in a single immutable object
- Immutable means this state object doesn't change
- a single source of truth
- managing the state: the Redux store.
- Redux object stores information about the all components state.

State



```
state
user:{
   name:"anonymous"
}
```

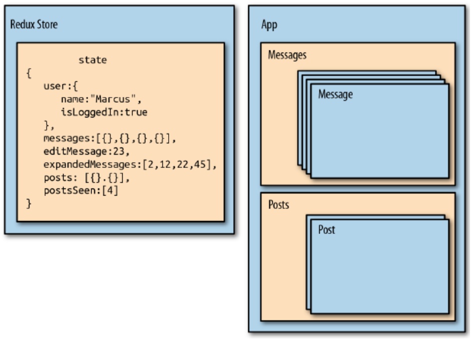
```
state
messages:[
   {...},
   {...},
   {...},
   {...}
]
```

```
state
editing: false,
expanded: true
```

```
state
posts:[
   {...},
   {...}
]
```

```
state
seen: false
```

State



```
state
{
   user:{
      name:"Marcus",
      isLoggedIn:true
   },
   messages:[{},{},{},{}],
   editMessage:23,
   expandedMessages:[2,12,22,45],
   posts: [{}.{}],
   postsSeen:[4]
}
```

Actions

- *Actions* provide: instructions about what should change in the application state along with the necessary data to make those changes
- Actions are the only way to update the state of a Redux application
- Actions provide us with instructions about what should change

Action Payload Data

- Most state changes also require some data called *payload*.
  - Which record should I remove?
  - What new information should I provide in a new record?

Reducer

- Redux achieves modularity via functions
- Functions are used to update parts of the state tree. These functions are called *reducers*
- Reducers are functions that take the current state along with an action as arguments and use them to create and return a new state
- Reducers are designed to update specific parts of the state tree, either leaves or branches.

The Store

- the store is what holds the application's state data and handles all state updates
- Flux design pattern allows for many stores that each focus on a specific set of data, Redux only has one store
- The store handles state updates by passing the current state and action through a single reducer
- create a store

```
import { createStore } from 'redux'
import { color } from './reducers'

const store = createStore(color)

console.log( store.getState() )
```

The Store

- Subscribing to Stores

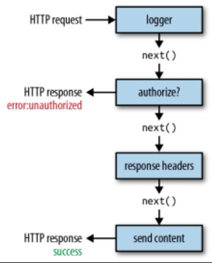
```
store.subscribe(() =>
    console.log('color count:', store.getState().colors.length)
)
```

- Saving to localStorage

```
const store = createStore(
    combineReducers({ colors, sort }),
    (localStorage['redux-store']) ?
        JSON.parse(localStorage['redux-store']) :
        {}
)
```

Middleware

- middleware serves as the glue between two different layers
- Middleware consists of a series of functions that are executed in a row in the process of dispatching an action



```
const logger = store => next => action => {
    let result
    console.groupCollapsed("dispatching", action.type)
    console.log('prev state', store.getState())
    console.log('action', action)
    result = next(action)
    console.log('next state', store.getState())
    console.groupEnd()
}
```

React Redux

- Explicitly Passing the Store

```
import React from 'react'
import ReactDOM from 'react-dom'
import App from './components/App'
import storeFactory from './store'

const store = storeFactory()

const render = () =>
    ReactDOM.render(
        <App store={store}/>,

         document.getElementById('react-container')
    )

store.subscribe(render)
render()
```

This is the *./index.js* file. In this file, we create the store with the storeFactory and
render the App component into the document

React Redux

- Explicitly Passing the Store

```
import AddColorForm from './AddColorForm'
import SortMenu from './SortMenu'
import ColorList from './ColorList'

const App = ({ store }) =>
    <div className="app">
        <SortMenu store={store} />
        <AddColorForm store={store} />
        <ColorList store={store} />
    </div>

export default App
```

passed the store to the App

continue to pass it down to
the child components that need it

React Redux

- Passing the Store via Context

```
import { PropTypes, Component } from 'react'
import SortMenu from './SortMenu'
import ColorList from './ColorList'
import AddColorForm from './AddColorForm'
import { sortFunction } from '../lib/array-helpers'

class App extends Component {

    getChildContext() {
        return {
            store: this.props.store
        }
    }

    componentWillMount() {
        this.unsubscribe = store.subscribe(
            () => this.forceUpdate()
        )
    }
```

For More details : https://reactjs.org/docs/context.html

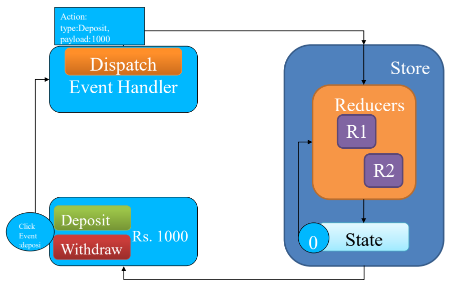
React Redux

```
componentWillUnmount() {
    this.unsubscribe()
}

render() {
    const { colors, sort } = store.getState()
    const sortedColors = [...colors].sort(sortFunction(sort))
    return (
        <div className="app">
            <SortMenu />
            <AddColorForm />
            <ColorList colors={sortedColors} />
        </div>
    )
}
}

App.propTypes = {
    store: PropTypes.object.isRequired
}

App.childContextTypes = {
    store: PropTypes.object.isRequired
}
                                                        export default App
```

React Redux

- STATE: Representing the global state of the App
- ACTION: Set of instruction, like handlers, it has two types of parameters {type,payload}
- REDUCERS: are functions, accept the current state + action and returns a new state.
- STORE: store contains reducers and you may access the global state of an app
- DISPATCH: the only way to update the state by calling store.dispatch.

React Redux



React Redux

- Presentational Components
  - Presentational components are components that only render UI elements
  - Are concerned with *how things look*
  - Often allow containment via `this.props.children`
  - Have no dependencies on the rest of the app, such as Flux actions or stores
  - Receive data and callbacks exclusively via props
- Container Components
  - Container components are components that connect presentational components to the data
  - Are concerned with *how things work*
  - Provide the data and behavior to presentational or other container components
  - Call Flux actions and provide these as callbacks to the presentational components
  - Are often stateful, as they tend to serve as data sources.

[Source: *https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0*]

React Redux

- The React Redux Provider
  - ease the complexity involved with implicitly passing the store via context
  - Dan Abramov, the creator of Redux
  - React Redux reduces your code's complexity and may help you build apps a bit faster
  - **`npm install react-redux --save`**
  - react-redux supplies us with a component that we can use to set up our store in the context, the *provider*

React Redux

```
import { Menu, NewColor, Colors } from './containers'

const App = () =>
    <div className="app">
        <Menu />
        <NewColor />
        <Colors />
    </div>

export default App
```

```
import React from 'react'
import { render } from 'react-dom'
import { Provider } from 'react-redux'
import App from './components/App'
import storeFactory from './store'

const store = storeFactory()

render(
    <Provider store={store}>
        <App />
    </Provider>,
    document.getElementById('react-container')
)
```

React Router

- Routing is the process of defining endpoints for your client's requests
- Incorporating the Router
  - npm install react-router-dom --save
  - <HashRouter> is changed to <BrowserRouter> and child component is <Routes>

```
render(
  <HashRouter>
    <div className="main">
        <Route exact path="/" component={Home} />
        <Route path="/about" component={About} />
        <Route path="/events" component={Events} />
        <Route path="/products" component={Products} />
        <Route path="/contact" component={Contact} />
    </div>
  </HashRouter>,
  document.getElementById('react-container')
)
```

React Router

- Router Properties

```
export const Whoops404 = ({ location }) =>
    <div className="whoops-404">
        <h1>Resource not found at '{location.pathname}'</h1>
    </div>
```

- <Link to>
- <Redirect to>

Nesting Routes

- Using a Page Template
- Subsections and Submenus
- Using redirects

Router Parameters

- Adding Color Details Page
- Multiple Parameters : Multiple parameters can be created and accessed on the same parameters object

```
<Route path="/members/:gender/:state/:city"
       component="Member" />


const Member = ({ match }) =>
    <div className="member">
        <ul>
            <li>gender: {match.params.gender}</li>
            <li>state: {match.params.state}</li>
            <li>city: {match.params.city}</li>
        </ul>
    </div>
```

JavaScript Object Notation (JSON)

- JavaScript Object Notation (JSON) is a standard text-based format for representing structured data based on JavaScript object syntax
- JSON is a text-based data format following JavaScript object syntax
- JSON structure: Key value pair
- Arrays as JSON

```
{
  "name": "Molecule Man",
  "age": 29,
  "secretIdentity": "Dan Jukes",
  "powers": [
    "Radiation resistance",
    "Turning tiny",
    "Radiation blast"
  ]
},
```

JavaScript Object Notation (JSON)

- JSON is a syntax for
  - serializing objects,
  - arrays,
  - numbers,
  - strings,
  - booleans,
  - null
- It is based upon JavaScript syntax but is distinct from it
- JSON.parse()

```
const json = '{"result":true, "count":42}';
const obj = JSON.parse(json);
```

- JSON.stringify()

```
console.log(JSON.stringify({ x: 5, y: 6 }));
// expected output: "{"x":5,"y":6}"
```

JavaScript Object Notation (JSON)

- Object basics
  - An object is a collection of related data and/or functionality

```
const person = {
  name: ['Bob', 'Smith'],
  age: 32,
  bio: function() {
    console.log(`${this.name[0]} ${this.name[1]} is ${this.age} years
old.`);
  },
  introduceSelf: function() {
    console.log(`Hi! I'm ${this.name[0]}.`);
  }
};
```

JavaScript Object Notation (JSON)

- Object basics
  - An object is a collection of related data and/or functionality

```
const person = {
  name: ['Bob', 'Smith'],
  age: 32,
  bio: function() {
    console.log(`${this.name[0]} ${this.name[1]} is ${this.age} years
old.`);
  },
  introduceSelf: function() {
    console.log(`Hi! I'm ${this.name[0]}.`);
  }
};
```

JavaScript Object Notation (JSON)

- 'this'
  - The this keyword refers to the current object the code is being written inside

```
const person1 = {
  name: 'Chris',
  introduceSelf() {
    console.log(`Hi! I'm ${this.name}.`);
  }
}
```

JavaScript Object Notation (JSON)

- constructor

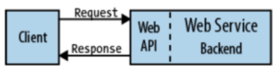
```
function createPerson(name) {
  const obj = {};
  obj.name = name;
  obj.introduceSelf = function() {
    console.log(`Hi! I'm ${this.name}.`);
  }
  return obj;
}
 const salva = createPerson('Salva');
 salva.name;
 salva.introduceSelf();
```

JavaScript Object Notation (JSON)

- JSON Schema
- https://rjsf-team.github.io/react-jsonschema-form/

REST API

- REpresentational State Transfer(REST) Application Programming Interfaces (API)
- Web services are purpose-built web servers that support the needs of a site or any other application
- Client programs use APIs to communicate with web services



REST API Design

- Designing a REST API must include:
  - When should URI path segments be named with plural nouns?
  - Which request method should be used to update resource state?
  - How do I map non-CRUD operations to my URIs?
  - What is the appropriate HTTP response status code for a given scenario?
  - How can I manage the versions of a resource's state representations?
  - How should I structure a hyperlink in JSON?
- Web Resource Modeling Language (WRML)
  - assist with the design and implementation of REST APIs
  - originated as a resource model diagramming technique
  - uses a set of basic shapes to represent each of the resource archetypes
  - WRML increased with the creation of the application/wrml media type
  - We have already discussed the structure of JSON Objects

Identifier Design with URIs

- URIs
  - REST APIs use Uniform Resource Identifiers (URIs) to address resources
- URI Format
  - URI = scheme "://" authority "/" path ["?" query] ["#" fragment]
- Rule: Forward slash separator (/) must be used to indicate a hierarchical relationship
- Rule: A trailing forward slash (/) should not be included in URIs
- Rule: Hyphens (-) should be used to improve the readability of URIs
- Rule: Underscores (_) should not be used in URIs
- Rule: Lowercase letters should be preferred in URI paths
- Rule: File extensions should not be included in URIs

Identifier Design with URIs

- URI Authority Design
- Rule: Consistent subdomain names should be used for your APIs
  - http://api.soccer.restapi.org
- Rule: Consistent subdomain names should be used for your client develop portal
  - http://developer.soccer.restapi.org

Identifier Design with URIs

- Resource Modeling
  - http://api.soccer.restapi.org/leagues/seattle/teams
  - http://api.soccer.restapi.org/leagues/seattle
  - http://api.soccer.restapi.org/leagues
  - http://api.soccer.restapi.org
- Resource Archetypes
  - Document: A document resource is a singular concept that is akin to an object instance or database record
  - Collection: A collection resource is a server-managed directory of resources
  - Store: A store is a client-managed resource repository
  - Controller: controller resource models a procedural concept

Identifier Design with URIs

- URI Query Design
  - URI = scheme "://" authority "/" path ["?" query] ["#" fragment]
- Rule: The query component of a URI may be used to filter collections or stores
- Rule: The query component of a URI should be used to paginate collection or store results

Interaction Design with HTTP

- HTTP/1.1:
  - REST APIs embrace all aspects of HTTP/1.1 including its
    - request methods, response codes, and message headers
- Request Methods
  - Rule: GET and POST must not be used to tunnel other request methods
  - Rule: GET must be used to retrieve a representation of a resource
  - Rule: HEAD should be used to retrieve response headers
  - Rule: PUT must be used to both insert and update a stored resource
  - Rule: PUT must be used to update mutable resources
  - Rule: POST must be used to create a new resource in a collection
  - Rule: POST must be used to execute controllers
  - Rule: DELETE must be used to remove a resource from its parent
  - Rule: OPTIONS should be used to retrieve metadata that describes a resource's available interactions

Interaction Design with HTTP

- Response Status Codes:
- 1xx: Informational Communicates transfer protocol-level information.
- 2xx: Success Indicates that the client's request was accepted successfully.
- 3xx: Redirection Indicates that the client must take some additional action in order to complete their request.
- 4xx: Client Error This category of error status codes points the finger at clients.
- 5xx: Server Error The server takes responsibility for these error status codes.

Interaction Design with HTTP

- Response Status Codes:
  - Rule: 200 ("OK") should be used to indicate nonspecific success
  - Rule: 200 ("OK") must not be used to communicate errors in the response body
  - Rule: 201 ("Created") must be used to indicate successful resource creation
  - Rule: 202 ("Accepted") must be used to indicate successful start of an asynchronous action
  - Rule: 204 ("No Content") should be used when the response body is intentionally empty
  - Rule: 301 ("Moved Permanently") should be used to relocate resources
  - Rule: 302 ("Found") should not be used
  - Rule: 303 ("See Other") should be used to refer the client to a different URI
  - Rule: 304 ("Not Modified") should be used to preserve bandwidth

Interaction Design with HTTP

- Response Status Codes:
  - Rule: 307 ("Temporary Redirect") should be used to tell clients to resubmit the request to another URI
  - Rule: 400 ("Bad Request") may be used to indicate nonspecific failure
  - Rule: 401 ("Unauthorized") must be used when there is a problem with the client's credentials
  - Rule: 403 ("Forbidden") should be used to forbid access regardless of authorization state
  - Rule: 404 ("Not Found") must be used when a client's URI cannot be mapped to a resource
  - Rule: 405 ("Method Not Allowed") must be used when the HTTP method is not supported
  - Rule: 500 ("Internal Server Error") should be used to indicate API malfunction

Representation Design

- Message Body Format:
  - Rule: JSON should be supported for resource representation
  - Rule: JSON must be well-formed
  - Rule: XML and other formats may optionally be used for resource representation
  - Rule: Additional envelopes must not be created
- Hypermedia Representation
  - Rule: A consistent form should be used to represent links
  - Rule: A consistent form should be used to represent link relations
  - Rule: A consistent form should be used to advertise links
  - Rule: Minimize the number of advertised "entry point" API URIs

Representation Design

- Media Type Representation:
  - Rule: A consistent form should be used to represent media type formats
  - Rule: A consistent form should be used to represent media type schemas
    - Schema Representation
    - Field Representation
    - Constraint Representation
    - Link Formula Representation
    - Document Schema Representation
    - Container Schema Representation
    - Collection Schema Representation
    - Store Schema Representation

Representation Design

- Error Representation:
  - Rule: A consistent form should be used to represent errors
  - Rule: A consistent form should be used to represent error responses
  - Rule: Consistent error types should be used for common error conditions

Cache

- Caching is one of web architecture's most important constraints.
- The cache constraints instruct a web server to declare the cacheability of each response's data.
- Caching response data can help to
  - reduce client-perceived latency,
  - increase the overall availability and reliability of an application, and
  - control a web server's load.
- In a word, caching reduces the overall cost of the Web.

Security

- Rule: OAuth may be used to protect resources
- Rule: API management solutions may be used to protect resources