# Frameworks Recap

## React, Grails, JavaScript, HTML, CSS, WTF

The point of our project is to learn, hand-on, about the software development process.

## React, Grails, JavaScript, HTML, CSS, WTF

Grails is hard to use.

Yes, it is, but it also does a lot of heavy lifting for you, and it plays nicely with Java. This is really the only reason I chose it over other frameworks.

## React, Grails, JavaScript, HTML, CSS, WTF

This isn’t a course about Grails, or about React.

Most of you indicated that you were interested in building a web-based application, rather than a desktop app.

The overall design of your application is still more important than the framework you use to implement it.

## REST Revisited

REST is a fundamental strategy for communication in web apps.

REST gives us a standard protocol for moving information between different programs written in different languages.

Major advantage

- One server for many client types, or
- One server for web, iOS, Android, etc.

## REST Revisited

Grails != REST

The basic purpose of REST is to expose application functionality to the outside world via URLs.

The client doesn’t need to know, or care what frameworks or programming languages are used to implement those functions.

Major advantages

- You can swap one REST server for another as long as they behave equivalently
- If Grails is just not working for you, you can re-implement your REST API using another framework (like Python/Flask)
- Your UI can’t tell the difference and it doesn’t care

## REST Revisited

To summarize:

- Grails can be used to implement a REST API.
- So can many other frameworks.
- User interfaces that *consume* a REST API don’t care how that API is implemented.
- REST APIs and user interfaces are interchangeable.

## So Why Grails?

- It uses Java, and you know Java
- It completely obviates the need for a separate database
- It’s actually simpler than other Java-based web frameworks
- It plays nicely with React, AngularJS, and HTML/CSS
- It has powerful authentication plugins
- It can implement a REST API out-of-the-box

## We’ve Talked About REST Before

… how does this actually help me?

If you take some time to understand REST, it will help you understand how to implement new features in React+Grails or any other framework.

Let’s use file uploading as an example.

## Uploading Files to a REST API

Disclaimer: This was a frustrating experience – until I calmed down and actually *thought* about what should be a sensible approach.

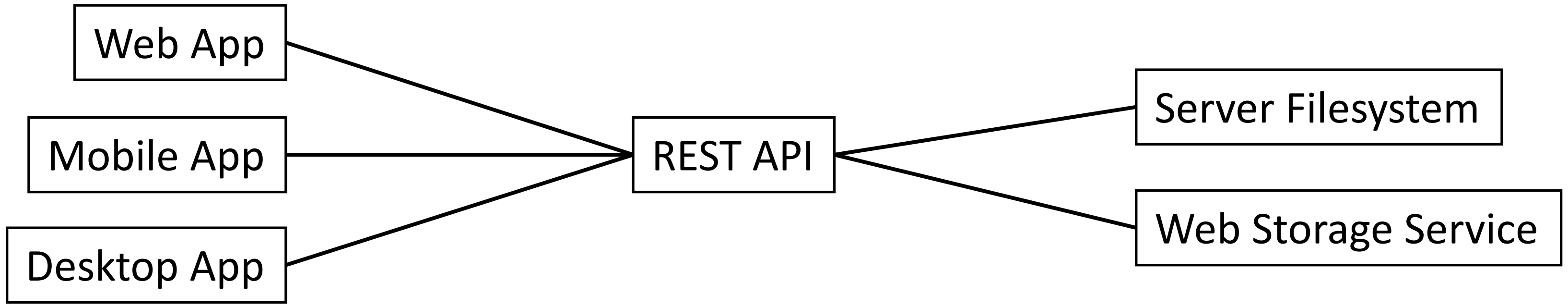
Unfortunately I was not able to find a single complete example of how to do this in Grails online.

Like some other things, the solution might be ***so obvious*** to experienced web programmers that it’s not worth mentioning in any tutorial.

## Uploading Files to a REST API

Problem:

Suppose we want to transfer a file from client to server, such as an image.



## Uploading Files to a REST API

So far, we have used GET to retrieve information and POST to send it.

In previous examples, we made POST calls by adding data directly to URL parameters.

```
fetch('http://localhost:8080/statusPost/postStatus?'
  + 'userName=' + name + "&statusText=" + status,
  {
    method: 'POST',
    headers: {
      "Content-Type": "application/json"
    }
  }
)
```

Example: Encoding data as POST request parameters.

## Uploading Files to a REST API

What if we want to send a 5 MB image to a REST API?

What strategies are available?

- Convert to binary, send corresponding string of 1’s and 0’s as URL parameters. Problems?
- Other formats?

## Uploading Files to a REST API

What if we want to send a 5 MB image to a REST API?

URLs were not designed to carry data payloads – especially large payloads. After all, URL stands for *Uniform Resource* ***Locator***.

We have already seen an alternative way of sending data as part of a REST API call in the authentication tutorial - authentication tokens are sent as part of the request ***header***.

## Uploading Files to a REST API

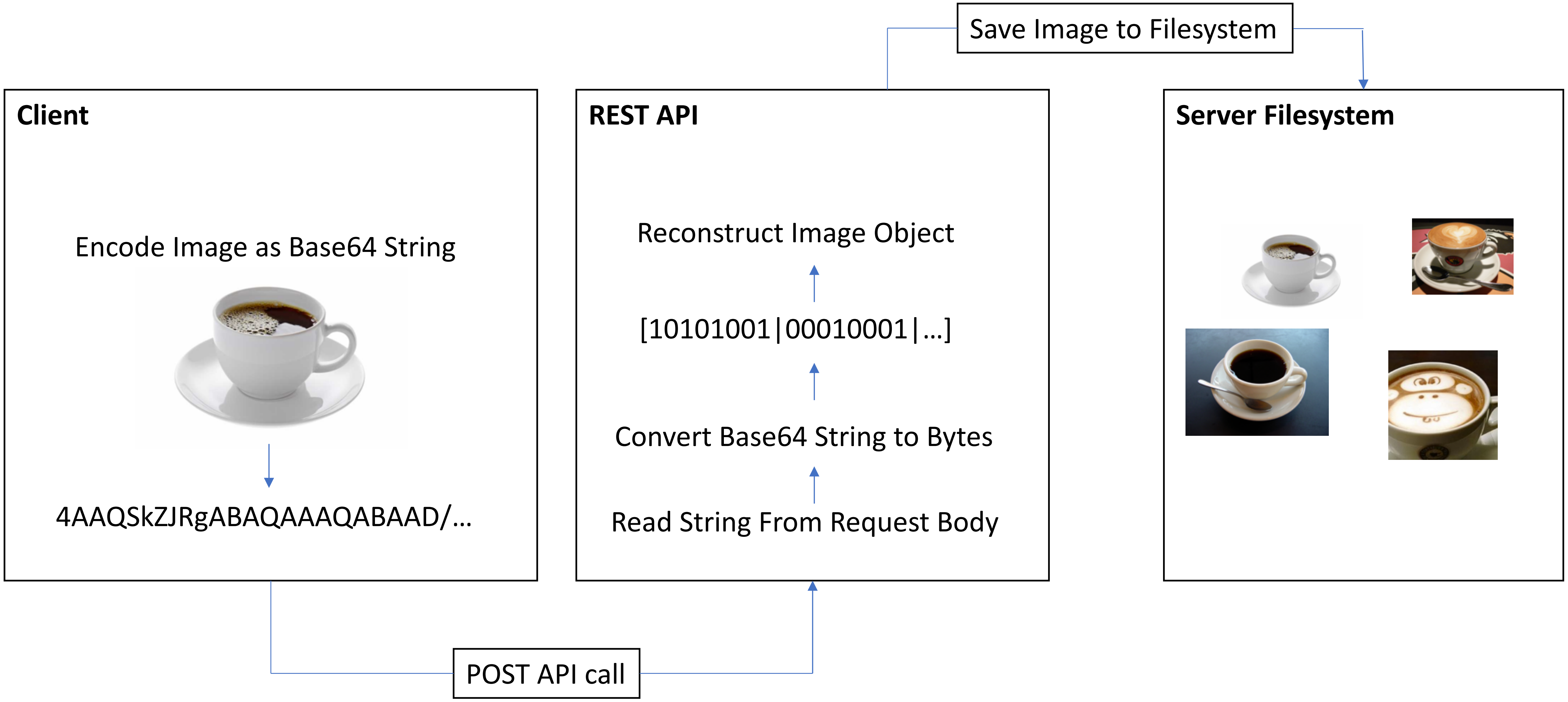
Another important part of a REST API call is the body.

A REST request body carries a data payload between client and server (or vice versa).

Though it is possible to send data encoded in other formats, it is simpler to use JSON across the board.

To send a file, however, this means we need to use an efficient text-based encoding.

# Uploading Files to a REST API


Save Image to Filesystem
Client
Encode Image as Base64 String
4AAQSkZJRgABAQAAAQABAAD/…
REST API
Reconstruct Image Object
[10101001|00010001|…]
Convert Base64 String to Bytes
Read String From Request Body
Server Filesystem
POST API call

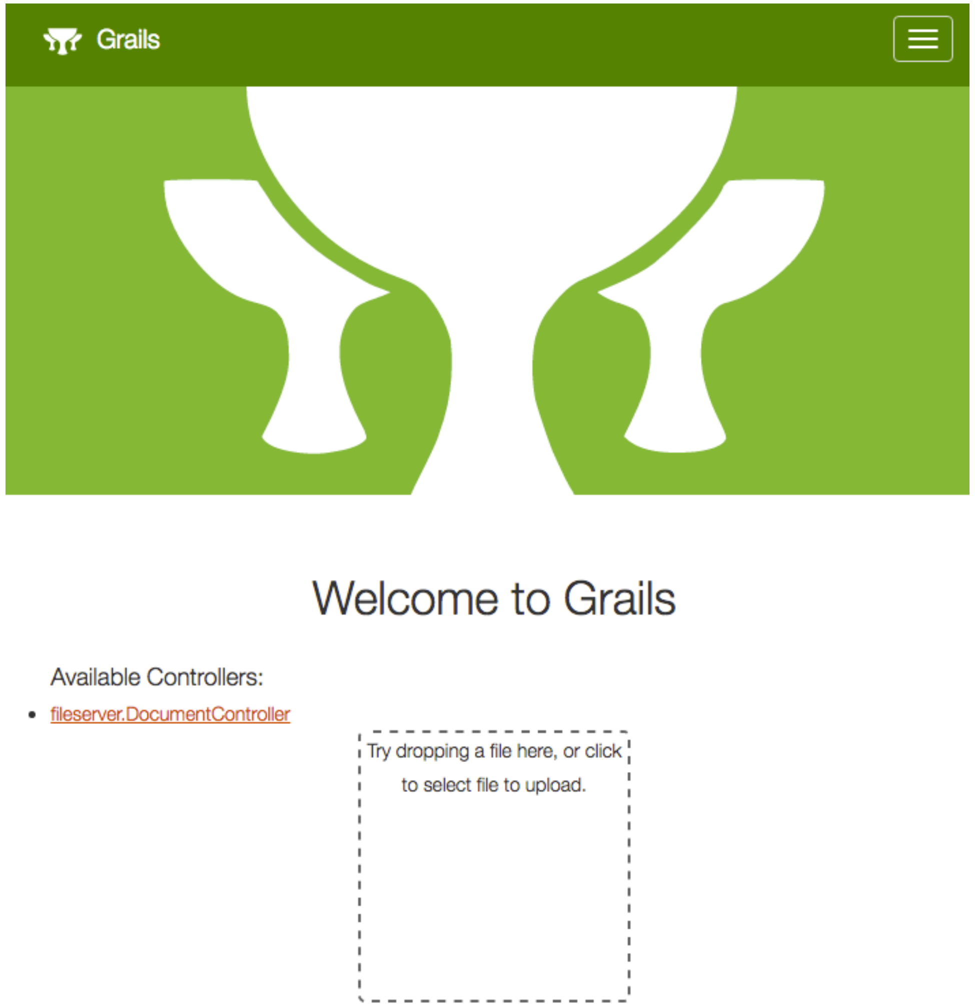
## Uploading Files to a REST API

Notes

- This general strategy should work with any framework and in any project that uses a REST API.
- An extra feature we might implement is to store the file data and metadata separately. (e.g. maintain a database of file *locations*, in addition to storing the files themselves.)

Let’s see how to do this in React+Grails.

# File Uploading in React



Welcome to Grails

Available Controllers:

- fileserver.DocumentController

Try dropping a file here, or click to select file to upload.

Let’s look at a very simple React use case – we’d like a drag-and-drop React component we can add to any HTML webpage.

Luckily, there is a free package called DropZone that does this for us. It can even be installed into your Grails project by using **npm**.

## File Uploading in React

File: ./grails-app/views/indes.gsp

```
…
<div id="app-div" align="center"></div>
<asset:javascript src="bundle.js"/>
…
```

Like our other React examples, all we need to do is create a div element in the webpage where we'd like to place a component.

This element will be added to the default page created when a Grails app is initialized.

# File Uploading in React

Now we need to

1) Modify the DropZone component to send a file to our API and
2) Render the component to our webpage.

For convenience, we can do all this in one JS file.

# File Uploading in React

```
function encodeAndUpload(file) {
    var reader = new FileReader();
    reader.readAsDataURL(file);
    reader.onload = function () {
        fetch('http://localhost:8080/document/upload?imageName='
          + file.name,{
            method: 'POST',
            headers: {
                "Content-Type": "application/json"
            },
            body: reader.result
        }).then(res =>{
            if(res.ok){
                alert("success!");
            }
            else{
                alert("fail!")
            }
        });
    };
}
```

/src/main/js/index.js

This code receives a file as a parameter and uses the FileReader class to convert it to Base64 encoding. This is the default encoding used by FileReader.readAsDataURL()

When the data is read, a typical fetch call is made to our API – we have a controller called UploadController waiting to receive data on the server side.

Notice that the body is simply being set as the encoded data, and that we are using URL parameters to send the metadata.

# File Uploading in React

```
var DropzoneDemo = React.createClass({
  onDrop: function (files) {
    console.log('Received files: ', files);
    encodeAndUpload(files[0])
  },

  render: function () {
    return (
      <div>
        <Dropzone onDrop={this.onDrop}>
          <div>Try dropping a file here, or click to select file to upload.</div>
        </Dropzone>
      </div>
    );
  }
});

ReactDOM.render(
  <div>
    <DropzoneDemo />
  </div>, document.getElementById('app-div'));
```

/src/main/js/index.js

The rest of the class is straightforward. It tells the DropZone component to call our encodeAndUpload method whenever it receives a newly dropped file.

Then, we simply render the component to its target webpage.

## Receiving Files in a Grails Controller

To receive a file in Grails, we need to implement a controller class that

1) Reads the file in Base64 format as the **body** of a POST request,
2) Converts the data back into an Image object, and
3) Stores the reconstructed file in the server’s filesystem

# Receiving Files in a Grails Controller

File: ./grails-app/controllers/DocumentController.groovy

```
class DocumentController extends RestfulController{

   static responseFormats = ['json', 'xml']
   static allowedMethods = [upload: 'POST']
```

First we need to set up a basic REST controller in Grails. We just need a single POST-enabled method to receive files.

# Receiving Files in a Grails Controller

File: ./grails-app/controllers/DocumentController.groovy

```
def upload() {
   try{
      decodeImage(request.inputStream.text, params.imageName)
      new Document(filename: params.imageName, fullPath:
            "./uploads/" + params.imageName).save()
      response.status = 200
   } catch (Exception e){
      e.printStackTrace()
      response.status = 500
   }
}
```

To receive the data, we just need to read the text data from the POST request body.

This contains our Base64 encoded file.

We can also create a metadata object and store it in the database.

## Decoding and Storing a File

Since we are receiving data encoded in Base64, we need to do some extra work to reconstruct the original file.

Luckily, Java provides lots of classes and methods we can use to work with various file formats, including images.

File: ./grails-app/controllers/DocumentController.groovy

```
// Helper method for decoding an image file
def decodeImage(String sourceData, String imageName){

   // tokenize the data
   def parts = sourceData.tokenize(",")
   def imageString = parts[1]

   // create a buffered image
   def image = null
   byte[] imageByte

   // decode from base64 back to bytes
   BASE64Decoder decoder = new BASE64Decoder()
   imageByte = decoder.decodeBuffer(imageString)
   ByteArrayInputStream bis = new ByteArrayInputStream(imageByte)

   // reconstruct an image from a byte array
   // if you are working with a different file format, you will need to use a class/method for that format
   image = ImageIO.read(bis)
   bis.close()

   // write the image to a file
   File outputfile = new File("./uploads/" + imageName)
   ImageIO.write(image, "png", outputfile)
}
```

data:image/jpeg;base64,/
9j/4AAQSkZJRgABAQAA
AQABAAD//

## Summary

Sending files over REST isn’t much more complicated than sending JSON or text data using headers and parameters.

It just requires means for:

1) The client to load the file
2) The client to encode the file
3) The server to decode the file
4) The server to store the file

All of these are basic features that can be implemented in any programming language.

## Summary

That being said – it can be difficult to find examples that combine your choice of front end and backend frameworks.

I was not able to find an example of file uploading with React+Grails, so I made one for your reference.

https://github.com/ethamajin/FileServer

More details can be found on the GitHub README.

## Project Plan

If you are getting nowhere with Grails, there is still time to move to another framework.

If you are implementing your backend as a REST API, then your frontend will work with any backend framework.

This is not ideal, but the option is on the table.

By using a REST API, you are not locking yourself into a single framework for your project. Both frontend and backend components can be changed out.

## Project Plan

Your Stage 2 is due on Friday.

Reasons you’re not getting an extension:

- The purpose of having an intermediate stage is to see how far you’ve come in this short time. Having an extra 2 days is just arbitrarily moving the target.
- The TAs need the weekend to review your progress so they can assign your Stage 3 Spec appropriately.
- Your Stage 3 Spec will be adjusted (hardened or softened) based on what you submit for Stage 2.
- You should really consider taking the weekend off…

## Tutorials, Quizzes

Though I’m being tough with deadlines for the project, I am opening up deadlines for everything else in the course.

**All tutorials and quizzes (past, present, and future) will ALL be due on March 31st at midnight – and not a minute later!**

Manage your time accordingly. Don’t miss out on easy marks.