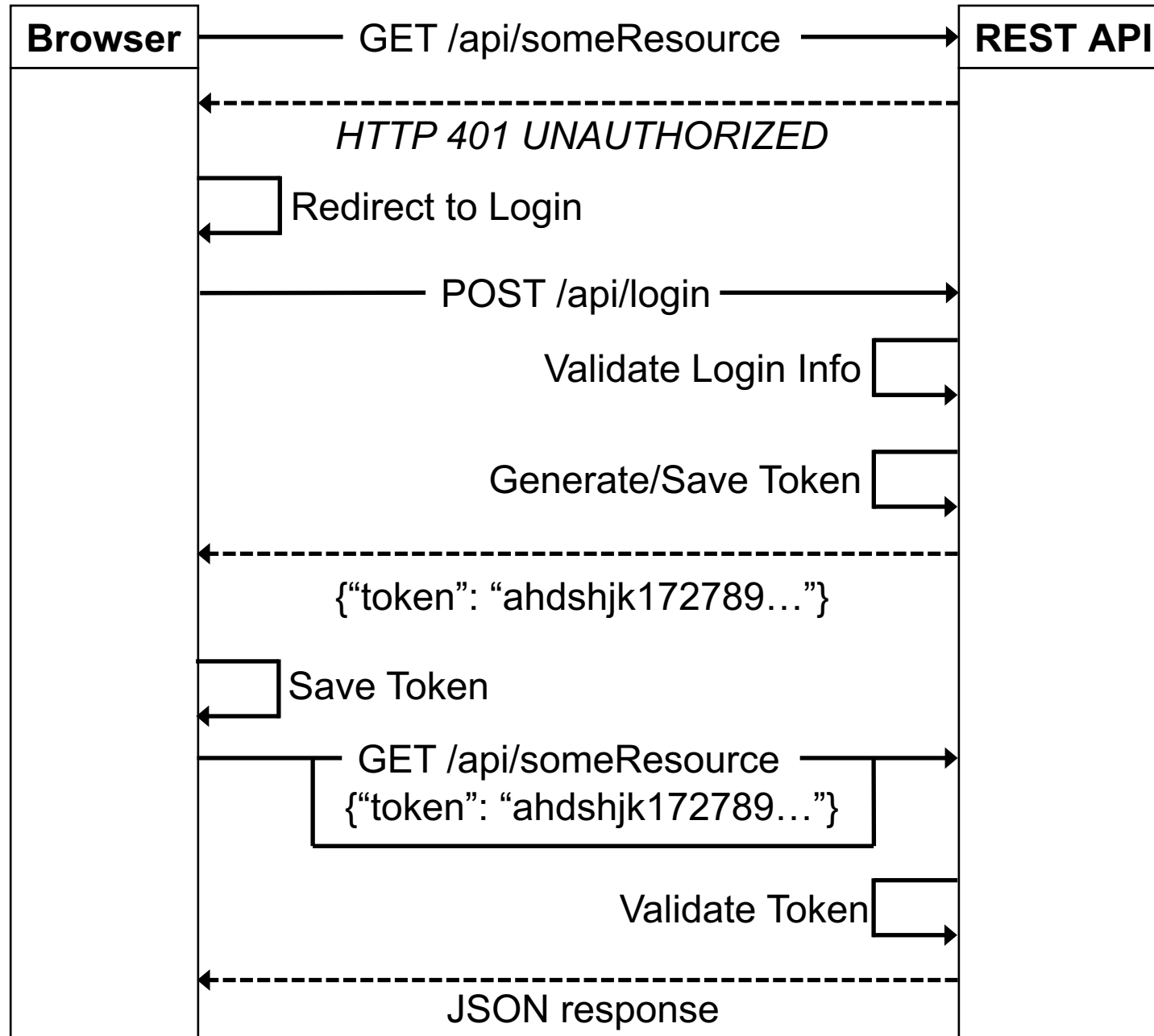# Authentication Tutorial

Authentication over REST

Motivation: one authentication strategy for all user interfaces.

We have already seen that a Grails-based REST controller supports multiple user interfaces.

Since we have already set up our controllers to communicate using REST calls, let's add authentication to this existing infrastructure.

Authentication over REST

Summary
- User attempts to access API resource requiring credentials
- Client is redirected to login form
- User enters credentials, UI sends to server using POST
- If valid, server generates and responds with token
- Client stores the token internally
- When user or client makes another API call, token will be sent
- Server uses the token for validation and permissions checking
- Server responds with requested resource or refusal code

REST Authentication in Grails

Grails applications can use the **spring-security-rest** plugin to help manage the process.

Documentation:
http://alvarosanchez.github.io/grails-spring-security-rest/latest/docs/index.html

There are several basic tutorials available including this pair:
http://sergiodelamo.es/how-to-use-spring-security-core-to-secure-you-grails-3-app/
http://sergiodelamo.es/how-to-secure-your-grails-3-api-with-spring-security-rest-for-grails/

Fully functional example available on GitHub
https://github.com/ethamajin/AuthenticationTutorial
(adapted from https://github.com/atapin/grails3-example)

REST Authentication in Grails

For the seriously interested, you should read through the resources mentioned in the previous slide.

It is recommended to play with a separate project while you experiment with authentication. After you understand the basic structure, you can start to integrate it into your project.

REST Authentication in Grails

The spring-security-rest plugin will create three domain classes by default: **User**, **Role**, and **UserRole**.

User will probably replace an existing class you had in mind for your project. Don't worry, you are free to modify this generated **User** class with your own features.

The remainder of this tutorial will focus on the example project.

Grails / REST / React Example Project

The example project was forked from
https://github.com/atapin/grails3-example

We will be using this example project for the next tutorial as well.

Like our earlier example project, it implements a RESTful web server and a React-based UI. The code is organized almost identically.

Example – Domain Classes

The domain classes in this example were generated automatically by calling:

grails s2-quickstart grails3.exampleUser Role

Important Features:
- Users have a **username** and **password**
- Roles have an **authority**
- UserRoles are used to assign a Role to a User

Example – Domain Classes

File: ./init/BootStrap

```
import grails3.example.Role
import grails3.example.User
import grails3.example.UserRole

class BootStrap {

    def init = { servletContext ->
        def role = new Role(authority: 'ROLE_USER').save()
        def user = new User(username: 'test', password: '2212').save()
        UserRole.create(user, role, true)
    }
    def destroy = {
    }
}
```

Initialization of an example Role, User and UserRole. You can test this account when you run the application.

Authority values are used to control access to **Controllers**, and **Controller Methods** using special annotations.

Example – Controller Classes

Two controller classes: **SearchController** and **UserController**.

Each controller has an associated **Service Class** in ./services.

Service classes contain logic that could otherwise be found in the controller class, make sense to separate as added functionality. For example, helper functions to make external API calls are appropriate for service classes.

Example – Controller Classes

File: SearchController.groovy

```groovy
@Secured(['ROLE_USER'])
class SearchController {

    static responseFormats = ['json']

    def searchService

    def search(String q) {
        log.debug("Searching by query = ${q}...")
        def result = searchService.search(q.trim())
        respond result
    }
    …
```

The @Secured annotation is a Spring Security annotation indicating that only Users with the authority **ROLE_USER** may use this controller.

Secured annotations make it easy to control which users have access to which controllers. You can use these annotations at the class level or at the method level for finer-detail control.

Example – Controller Classes

File: SearchController.groovy

```groovy
@Secured(['ROLE_USER'])
class SearchController {

    static responseFormats = ['json']

    def searchService

    def search(String q) {
        log.debug("Searching by query = ${q}...")
        def result = searchService.search(q.trim())
        respond result
    }
    …
```

def searchService loads in an instance of the service class. It's just a normal Java-like class that sets up an API call to Twitter. We will see this in the next tutorial, but feel free to explore in the meantime.

Notice the use of a parameter to the search method. We could equivalently use **params.q** instead of specifying the parameter directly.

Otherwise, this is just a typical controller.

Example – Controller Classes

File: UserController.groovy

```groovy
class UserController {

    static responseFormats = ['json']

    def userService

    def signUp(@RequestParameter('username') String username,
               @RequestParameter('password') String password) {
        log.debug("Signing up a new user: ${username}:[*****]")
        def user = userService.signUp(username, password)
        def payload = [username: user.username] as Object
        respond payload, status: HttpStatus.CREATED
    }
}
```

Similar to the last example, **UserController** also uses a service class.

The signUp method demonstrates yet another syntax to accepting parameters from URLs.

This is also equivalent to taking in values from **params.username** and **params.password**.

Example – Controller Classes

A login controller was automatically defined by **spring-security-rest**.
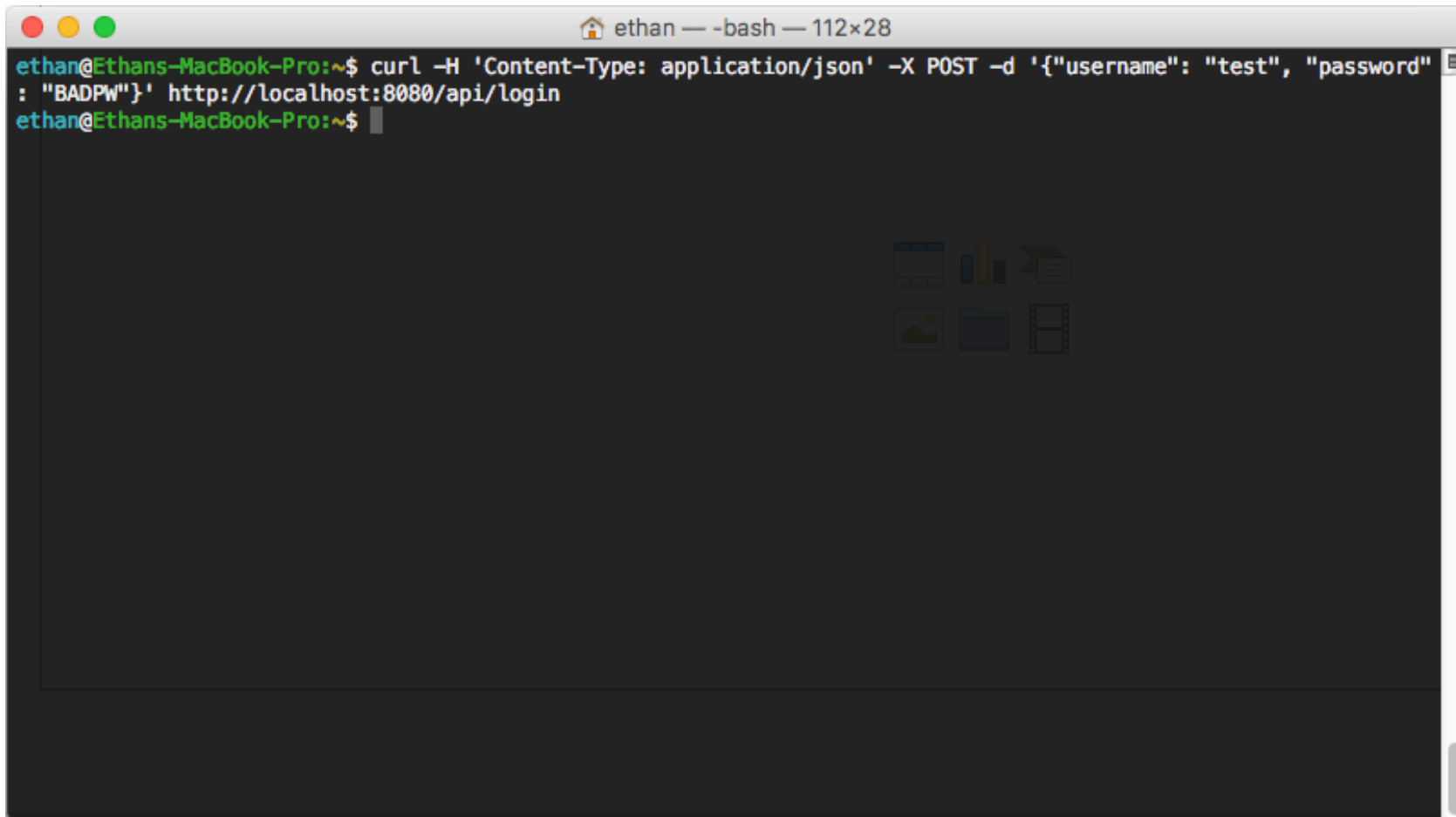
We can test it by making an API call.

Testing the login API

```
curl -H 'Content-Type: application/json' -X POST -d '{"username": "test", "password": "2212"}'
http://localhost:8080/api/login
```

Testing the login API

```
curl -H 'Content-Type: application/json' -X POST -d '{"username": "test", "password": "BADPW"}' http://localhost:8080/api/login
```

Testing the login API

The response from the first request was successful.

It contained:
- The username
- The user's authorized roles
- An access token (expire, within a reasonable session duration)
- A refresh token (used to request new access tokent)

For now, all we need to know is that Spring Security is managing tokens for us, and we can use them transparently in our application.

Testing the login API

Questions:
How secure is this system?
Can you identify any possible flaws?

Tell Grails to run in HTTPS mode by modifying the run configuration to **grails run-app -https**

Example – React Views

So far, we have configured **SearchController** to only respond to requests coming from authenticated users with the **ROLE_USER** authority.

We then tested the login API from the command line.

Now let's see how to log in from a React UI.

Example – React Views

File: signin.js

```javascript
signIn(e) {
    e.preventDefault();
    console.log("Signing in...", this.form.data());

    fetch("/api/login", {
        method: 'POST',
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(this.form.data())
    })
    .then(checkStatus)
    .then(this.success.bind(this))
    .catch(this.fail.bind(this))
}
```

This React component is responsible for making the API call to our login controller.

The username and password are being wrapped up into a JSON object and sent as the POST call body.

The request is then identical to the one we sent using the command line.

Example – React Views

File: signin.js

```javascript
signIn(e) {
    e.preventDefault();
    console.log("Signing in...", this.form.data());

    fetch("/api/login", {
        method: 'POST',
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(this.form.data())
    })
    .then(checkStatus)
    .then(this.success.bind(this))
    .catch(this.fail.bind(this))
}
```

The .then() methods will execute once the server responds.

checkStatus is another function in the same class that interprets the response code.

If the response was successful, then the second .then will bind the JSON response – exactly what we saw on the command line – to an internal storage mechanism so that the tokens are available in the session. *See auth.js for details.*

Example – React Views

File: search.js

```
search(e) {
  e...
  c
  le
  le                                    trim();
  t
  f
  }
  .then(checkStatus)
  .then(this.success)
  .catch(this.fail)
}
```

```groovy
@Secured(['ROLE_USER'])
class SearchController {

    static responseFormats = ['json']

    def searchService

    def search(String q) {
        log.debug("Searching by query = ${q}...")
        def result = searchService.search(q.trim())
        respond result
    }
    …
```

This function is called when the search form is submitted.

This class is initialized with the token as a state variable, and simply includes it as part of the API call header.

Recall the implementation of SearchController on the client side.

Aside from the @Secured annotation, our authorization protocol is almost entirely transparent!

Example – React Views

The remainder of the UI components in the example code are outside the scope of this tutorial.

For teams interested in using React, I recommend the following tutorial for more advanced features including those used in the example code. It assumes just the basic knowledge you should have from our tutorial on React.

https://css-tricks.com/learning-react-router/

Example – Information from Tokens

To control permissions, we may want to extract information about the current user given an authenticated REST call.

For example, we might use the username field to determine whether access should be granted to a secret event group.

To do this, we can use a built-in service class called SpringSecurityService.

Example – Information from Tokens

File: SearchController.groovy

```groovy
@Secured(['ROLE_USER'])
class SearchController {

    static responseFormats = ['json']

    def searchService
    def springSecurityService

    def search(String q) {
        // Gets the current user name - can be used to control permissions
        def info = springSecurityService.currentUser.username
        log.debug("Searching by query = ${q}...")
        def result = searchService.search(q.trim())
        respond result
    }
}
```

Adding a reference called springSecurityService enables the extraction of user information from an authenticated API call.

Again, note the transparancy of this approach. We are never actually dealing with permission levels or tokens manually – we can set up a very simple system to do this for us!

Notes on Security

We have only scratched the surface of securing our Grails application.

This example provides some basic security, but it is a much more complex issue than we can cover.

I strongly recommend reading all the resources linked in these slides.

Notes on Security

For your projects, basic security and authentication is important.

However, I don't expect anyone to become a security expert in six weeks – and our expectations in evaluations will reflect this.

Where appropriate, though, you should attempt to implement a model similar to the example.

You can even use this project as a starter.

Your Tutorial Task

Enjoy Reading Week!