CAS Internals

A Journey Through the CAS Codebase

```
@FunctionalInterface
public interface AuthenticationManager {
  String BEAN_NAME = "casAuthenticationManager";

Authentication authenticate(AuthenticationTransaction
authenticationTransaction)
  throws Throwable;
}
```

Dmitriy Kopylenko

Contents

CAS Internals: A Journey Through the Central Authentication Server Codebase

0.1 About This Book

This book is for developers who want to understand how Apereo CAS (Central Authentication Service) is actually implemented. Not configuration guides or deployment tutorials—this is a deep dive into the source code, architecture decisions, and implementation patterns that make CAS work.

0.2 Who This Is For

- **CAS contributors and maintainers** who need to understand the codebase deeply
- **Developers building similar authentication systems** who want to learn from a mature implementation
- **Architects evaluating CAS** for enterprise use who need to understand what's under the hood
- **Java developers** wanting to study a complex, real-world Spring Boot application
- **Anyone debugging, extending, or optimizing CAS** at the code level

0.3 What You'll Learn

- How CAS bootstraps and wires together hundreds of components
- Authentication flow internals and Spring Webflow state machine mechanics
- Ticket system implementation: creation, storage, expiration, and cleanup
- Service registry architecture and attribute release engine
- Protocol engines (SAML2, OAuth2, OIDC) implementation details
- Security patterns: encryption, signing, JWT handling
- Performance optimization: virtual threads, caching, connection pooling
- Testing strategies and extension points

- How to navigate and contribute to the massive CAS codebase

0.4 What This Book Is NOT

- Not a user guide or configuration reference
- Not a deployment tutorial
- Not protocol specification documentation
- Not focused on specific authentication providers (LDAP, JDBC, etc.)

This is about **how CAS is built**, not how to use it.

0.5 Prerequisites

- **Strong Java knowledge** (generics, lambdas, streams, concurrency)
- **Spring Framework experience** (IoC, auto-configuration, aspects)
- **Spring Boot understanding** (starters, auto-configuration, properties)
- **Basic authentication concepts** (sessions, cookies, tokens, SSO)
- **Ability to read large codebases** and navigate package structures

0.6 About the Authors

Dmitriy Kopylenko is an IAM specialist with deep expertise in Apereo CAS, Shibboleth IdP, SAML, OAuth2, and OIDC protocols. He has been a long-time committer to CAS and was involved with CAS since the original CAS 3.0 architecture and implementation.

0.7 Book Version & Reference

This book uses **CAS 7.3** as the reference implementation for all examples and code analysis.

- **CAS Version:** 7.3
- **Last Updated:** March 2026
- **Repository:** github.com/apereo/cas
- **Java Version:** 21+ (with virtual threads)
- **Spring Boot Version:** 3.x

0.8 How to Use This Book

Each chapter is self-contained but builds on previous concepts. You can:

- **Read sequentially** for comprehensive understanding
- **Jump to specific topics** using the detailed table of contents
- **Use as reference** when debugging or extending CAS
- **Follow along with the code** - all examples reference actual CAS source files

0.9 Conventions Used

- **Code paths** are relative to CAS repository root
- **Package names** use abbreviated format when clear from context
- **Code snippets** are extracted from actual CAS 7.3 source
- **Line numbers** may shift between minor versions
- **Key classes** are **bolded** when first introduced

0.10 Typographic Conventions

- `ClassName` - Java classes and interfaces
- `methodName()` - Java methods
- `propertyName` - Configuration properties
- *emphasis* - Important concepts
- **strong emphasis** - Critical information

0.11 Feedback and Updates

This book is continuously updated as CAS evolves. Your feedback helps improve it.

- **Issues/Suggestions:** Submit via the Leanpub book page or email the author directly
- **Updates:** Published regularly on Leanpub
- **Errata:** Tracked on the Leanpub book page

0.12 Acknowledgments

Thanks to the Apereo CAS community and the core maintainers who have built and documented this incredible system. Special thanks to Misagh Moayyed and the CAS team for their tireless work.

0.13 Let's Dive In

Ready to understand how CAS really works? Let's start with the architecture…

1

Architecture & Design Philosophy

1.1 Overview

CAS is not just an authentication server—it's a massive, modular Spring Boot ecosystem with nearly 400 separate modules working together through carefully crafted abstractions and auto-configuration. Understanding this architecture is essential before diving into specific features.

This chapter reveals how CAS is structured, why certain design decisions were made, and how all these components wire themselves together at runtime. You'll see the patterns repeated throughout the codebase and understand the philosophy that guides CAS development.

By the end of this chapter, you'll understand the big picture: how a simple `java -jar cas.war` command bootstraps hundreds of beans, wires complex authentication flows, and creates a production-ready authentication server.

1.2 Prerequisites

- Spring Framework fundamentals (IoC, dependency injection, bean lifecycle)
- Spring Boot basics (auto-configuration, starters, conditional beans)
- Java package organization and module systems
- Basic understanding of SSO and authentication concepts

1.3 Key Concepts

- **Modular Architecture**: CAS is divided into ~400 modules, each with a specific responsibility

- **Auto-Configuration**: Spring Boot's conditional configuration drives component assembly
- **Layered Abstraction**: API -> Core -> Support modules create clear separation of concerns
- **Convention over Configuration**: Sensible defaults with override capabilities
- **Pluggable Components**: Nearly everything can be replaced via Spring beans

---

1.4 The Big Picture: What is CAS?

CAS as a Service-Oriented System

At its core, CAS is defined by a single interface that has remained conceptually stable since CAS 3.0:

```java
// File:
//   api/cas-server-core-api/.../CentralAuthenticationService.java

public interface CentralAuthenticationService {
    Ticket createTicketGrantingTicket(AuthenticationResult authenticationResult)
 ↪  throws Throwable;

    Ticket grantServiceTicket(String ticketGrantingTicketId, Service service,
                              AuthenticationResult authenticationResult) throws
 ↪  Throwable;

    Assertion validateServiceTicket(String serviceTicketId, Service service)
 ↪  throws Throwable;

    Ticket createProxyGrantingTicket(String serviceTicketId,
                                    AuthenticationResult authenticationResult)
 ↪  throws Throwable;

    Ticket grantProxyTicket(String proxyGrantingTicket, Service service);

    TicketFactory getTicketFactory();
}
```

This interface tells you everything about CAS's purpose:

- **Create tickets** (TGT, ST, PGT, PT) representing authentication state
- **Validate tickets** to assert user identity
- **Grant service access** through ticket-based workflows

Everything else in CAS—SAML, OAuth, OIDC, MFA, delegated auth—builds on this foundation.

Why This Design:

The interface-first approach allows CAS to:

1. Support multiple protocol implementations (CAS, SAML, OAuth, OIDC) using the same ticket engine
2. Swap implementations (ticket storage, authentication mechanisms) without changing contracts
3. Test components in isolation using mocks
4. Maintain backward compatibility while evolving internals

Runtime Behavior:

When a user authenticates, CAS:

1. Creates a `TicketGrantingTicket` (session)
2. Issues `ServiceTicket` instances for each service access
3. Validates tickets and returns `Assertion` objects with principal attributes
4. Manages ticket lifecycle (expiration, renewal, deletion)

1.5 Module Organization: The Three-Layer Architecture

CAS is organized into three fundamental layers:

1. API Layer (`api/`)

Purpose: Define contracts (interfaces, enums, exceptions) with zero implementation

```
api/
├── cas-server-core-api/                    # Core CAS interfaces
├── cas-server-core-api-audit/              # Audit abstractions
├── cas-server-core-api-authentication/     # Authentication contracts
├── cas-server-core-api-configuration/      # Configuration metadata
├── cas-server-core-api-configuration-model/ # Configuration property model
├── cas-server-core-api-cookie/             # Cookie management contracts
├── cas-server-core-api-events/             # Event system abstractions
├── cas-server-core-api-logout/             # Logout interfaces
├── cas-server-core-api-mfa/                # MFA framework contracts
├── cas-server-core-api-monitor/            # Monitoring abstractions
├── cas-server-core-api-multitenancy/       # Multitenancy contracts
├── cas-server-core-api-protocol/           # Protocol abstractions
├── cas-server-core-api-scripting/          # Scripting abstractions
├── cas-server-core-api-services/           # Service registry contracts
├── cas-server-core-api-throttle/           # Throttling contracts
├── cas-server-core-api-ticket/             # Ticket system interfaces
├── cas-server-core-api-util/               # Utility interfaces
├── cas-server-core-api-validation/         # Validation abstractions
├── cas-server-core-api-web/                # Web abstractions
└── cas-server-core-api-webflow/            # Webflow contracts
```

Key Characteristics:

- No dependencies on implementations
- Minimal transitive dependencies
- Used by all other layers
- Changes here affect the entire system

2. Core Layer (`core/`)

Purpose: Default implementations of API contracts and essential functionality

Statistics: 45 core modules providing foundational capabilities

```
core/
├── cas-server-core/                       # Main CAS implementation
├── cas-server-core-audit/                 # Audit implementation
├── cas-server-core-authentication/        # Authentication engine
├── cas-server-core-authentication-api/    # Authentication API extensions
├── cas-server-core-authentication-attributes/ # Attribute resolution
├── cas-server-core-authentication-mfa/    # MFA framework core
├── cas-server-core-configuration/         # Configuration binding
├── cas-server-core-cookie/              # Cookie handling
├── cas-server-core-events/              # Event system
├── cas-server-core-logout/              # Logout implementation
├── cas-server-core-notifications/       # Notifications engine
├── cas-server-core-scripting/           # Scripting support
├── cas-server-core-services/            # Service registry core
├── cas-server-core-tickets/             # Ticket implementation
├── cas-server-core-util/                # Core utilities
├── cas-server-core-validation/          # Validation engine
├── cas-server-core-web/                 # Web infrastructure
├── cas-server-core-webflow/             # Webflow implementation
└── ... (28 more modules)
```

Design Pattern: Each core module typically contains:

1. **Implementations** of API interfaces
2. **Configuration classes** (`@AutoConfiguration` beans)
3. **Support classes** (builders, factories, helpers)
4. **Tests** (extensive unit and integration tests)

3. Support Layer (`support/`)

Purpose: Specific integrations, protocol implementations, and optional features

Statistics: 348 support modules providing extensibility

Organized by feature category:

```
support/
├── Ticket Registries (storage backends)
│   ├── cas-server-support-redis-ticket-registry/
│   ├── cas-server-support-jpa-ticket-registry/
│   ├── cas-server-support-hazelcast-ticket-registry/
│   ├── cas-server-support-memcached-ticket-registry/
│   └── ... (10+ ticket registry implementations)
│
├── Authentication Methods
│   ├── cas-server-support-ldap/
│   ├── cas-server-support-jdbc/
│   ├── cas-server-support-radius/
│   ├── cas-server-support-spnego/
│   └── ... (30+ authentication providers)
│
```

```
├── MFA Providers
│   ├── cas-server-support-duo/
│   ├── cas-server-support-yubikey/
│   ├── cas-server-support-google-authenticator/
│   └── ... (15+ MFA implementations)
│
├── Protocol Implementations
│   ├── cas-server-support-saml-idp/
│   ├── cas-server-support-oauth/
│   ├── cas-server-support-oidc/
│   ├── cas-server-support-ws-idp/
│   └── ... (protocol engines)
│
├── Service Registries
│   ├── cas-server-support-jpa-service-registry/
│   ├── cas-server-support-mongo-service-registry/
│   ├── cas-server-support-ldap-service-registry/
│   └── ... (12+ service registry backends)
│
└── ... (dozens of other categories)
```

Why This Organization:

1. **Clear dependencies**: API <- Core <- Support (no circular dependencies)
2. **Selective deployment**: Include only needed support modules
3. **Independent evolution**: Support modules can be updated independently
4. **Testing boundaries**: Each layer can be tested in isolation
5. **Gradle subprojects**: Each module is a separate Gradle project

---

1.6 Spring Boot Bootstrap Process

The Entry Point

```
// File:
//   webapp/cas-server-webapp-init/.../web/CasWebApplication.java

@EnableDiscoveryClient
@SpringBootApplication(proxyBeanMethods = false,
    exclude = {
        DataSourceAutoConfiguration.class,
        HibernateJpaAutoConfiguration.class,
        MailSenderAutoConfiguration.class,
        MongoAutoConfiguration.class,
        MongoDataAutoConfiguration.class
    })
@EnableConfigurationProperties(CasConfigurationProperties.class)
@EnableAspectJAutoProxy(proxyTargetClass = false)
@EnableTransactionManagement(proxyTargetClass = false)
@EnableScheduling
@EnableAsync(proxyTargetClass = false)
@NoArgsConstructor
public class CasWebApplication {

    public static void main(final String[] args) {
        val applicationClasses = getApplicationSources(args);
        new SpringApplicationBuilder()
            .sources(applicationClasses.toArray(ArrayUtils.EMPTY_CLASS_ARRAY))
            .banner(CasBanner.getInstance())
            .web(WebApplicationType.SERVLET)
            .logStartupInfo(true)
            .applicationStartup(ApplicationUtils.getApplicationStartup())
            .run(args);
    }

    protected static List<Class> getApplicationSources(final String[] args) {
        val applicationClasses = new ArrayList<Class>();
        applicationClasses.add(CasWebApplication.class);
        ApplicationUtils.getApplicationEntrypointInitializers()
            .forEach(init -> {
                init.initialize(args);
                applicationClasses.addAll(init.getApplicationSources(args));
            });
        return applicationClasses;
    }
}
```

Key Observations:

1. `@SpringBootApplication`: Triggers component scanning and auto-configuration
2. **Exclusions**: CAS disables default Spring Boot auto-configurations it doesn't need
3. `proxyBeanMethods = false`: Optimization to avoid CGLIB proxying (faster startup)

4. **Multiple enablement annotations**: AOP, transactions, scheduling, async processing
5. **Dynamic source loading**: `ApplicationEntrypointInitializers` allow plugins to contribute configuration

Why Exclude Default Auto-Configurations?

CAS excludes `DataSourceAutoConfiguration`, `HibernateJpaAutoConfiguration`, etc., because:

- CAS conditionally creates these beans only when needed
- Prevents unwanted datasource initialization at startup
- Allows multiple datasource configurations (tickets, service registry, audit)
- Gives CAS fine-grained control over component lifecycle

The Bootstrap Sequence:

1. **JVM starts** -> `main()` method invoked
2. **SpringApplicationBuilder** created with CasWebApplication as primary source
3. **ApplicationEntrypointInitializers** discovered via ServiceLoader
4. **Additional configuration classes** added dynamically
5. **Spring Boot runs** -> Component scan starts
6. **Auto-configuration classes** processed based on conditions
7. **Beans instantiated** in dependency order
8. **Application context refreshed** -> CAS ready to serve requests

---

1.7 The Auto-Configuration Pattern

CAS uses Spring Boot's auto-configuration extensively. Each module typically provides a configuration class:

Auto-Configuration Class Structure

```java
// Example pattern from cas-server-core-notifications

@AutoConfiguration
@EnableConfigurationProperties(CasConfigurationProperties.class)
@ConditionalOnFeatureEnabled(feature =
↪   CasFeatureModule.FeatureCatalog.Notifications)
public class CasCoreNotificationsAutoConfiguration {

    @Bean
    @ConditionalOnMissingBean(name = CommunicationsManager.BEAN_NAME)
    @RefreshScope(proxyMode = ScopedProxyMode.DEFAULT)
    public CommunicationsManager communicationsManager(
            @Qualifier(SmsSender.BEAN_NAME) final SmsSender smsSender,
            @Qualifier(EmailSender.BEAN_NAME) final EmailSender emailSender,
            @Qualifier(NotificationSender.BEAN_NAME) final NotificationSender
            ↪   notificationSender) {
        return new DefaultCommunicationsManager(smsSender, emailSender,
        ↪   notificationSender);
    }

    @Bean
    @ConditionalOnMissingBean(name = EmailSender.BEAN_NAME)
    @RefreshScope(proxyMode = ScopedProxyMode.DEFAULT)
    public EmailSender emailSender(
            final CasConfigurationProperties casProperties) {
        return new DefaultEmailSender(casProperties.getEmailProvider());
    }

    // ... more beans
}
```

Key Patterns:

1. **`@AutoConfiguration`**: Marks this as an auto-configuration class
2. **`@EnableConfigurationProperties`**: Binds `cas.*` properties to POJOs
3. **`@ConditionalOnFeatureEnabled`**: CAS-specific annotation that gates entire configuration classes behind feature flags
4. **`@ConditionalOnMissingBean`**: Only create if user hasn't provided custom bean
5. **Qualifier annotations**: Explicitly name beans to avoid conflicts
6. **Property injection**: Configuration properties drive behavior

Conditional Bean Creation:

CAS extensively uses Spring Boot conditionals:

- `@ConditionalOnMissingBean(name = "...")`: Create only if no custom implementation
- `@ConditionalOnFeatureEnabled(feature = ...)`: CAS-specific gate for entire feature modules
- `@ConditionalOnProperty(name = "cas.feature.enabled")`: Create only if enabled
- `@ConditionalOnClass(Foo.class)`: Create only if dependency is present
- `@ConditionalOnBean(Bar.class)`: Create only if another bean exists

Why This Approach?

1. **Extensibility**: Override any bean by defining your own with the same name
2. **Modularity**: Each module contributes beans independently
3. **No XML**: Pure Java configuration
4. **Testability**: Easy to provide test doubles
5. **Documentation**: Code is self-documenting

1.8 Configuration Property Binding

The CasConfigurationProperties Mega-Class

CAS centralizes all configuration in a single nested properties class:

```
@EnableConfigurationProperties(CasConfigurationProperties.class)
```

This binds all `cas.*` properties to a Java object graph:

```
cas.authn.ldap[0].ldapUrl=ldap://localhost:389
cas.authn.ldap[0].baseDn=dc=example,dc=org
cas.ticket.registry.redis.host=localhost
cas.ticket.registry.redis.port=6379
```

Architecture:

```
CasConfigurationProperties
├── AuthnProperties authn
│   ├── LdapProperties[] ldap
│   ├── JdbcProperties jdbc
│   └── ... (50+ authentication mechanisms)
├── TicketProperties ticket
│   ├── RegistryProperties registry
│   │   ├── RedisProperties redis
│   │   ├── JpaProperties jpa
│   │   └── ... (ticket storage options)
│   └── ExpirationProperties expiration
├── ServiceRegistryProperties serviceRegistry
├── MonitorProperties monitor
└── ... (hundreds of nested property classes)
```

Benefits:

1. **Type safety**: Properties are strongly typed POJOs
2. **Validation**: Use JSR-303 annotations (`@NotNull`, `@Pattern`, etc.)
3. **IDE support**: Auto-completion in application.properties
4. **Documentation**: Generate property documentation from code
5. **Defaults**: Embedded in POJO initialization

1.9 Design Patterns Used Throughout CAS

CAS employs consistent patterns across all modules:

1. Strategy Pattern (Ticket Registries)

```java
public interface TicketRegistry {
    Ticket addTicket(Ticket ticket) throws Exception;
    Ticket getTicket(String ticketId);
    Collection<? extends Ticket> getTickets();
    int deleteTicket(String ticketId) throws Exception;
    Ticket updateTicket(Ticket ticket) throws Exception;
    // ...
}

// Implementations:
// - JpaTicketRegistry
// - RedisTicketRegistry
// - HazelcastTicketRegistry
// - MemcachedTicketRegistry
// - (20+ implementations)
```

All implementations are interchangeable. Choose at deployment time via properties.

2. Factory Pattern (Ticket Creation)

```
public interface TicketFactory {
    TicketFactory get(Class<? extends Ticket> clazz);
}

// Creates tickets polymorphically
// Each ticket type has a dedicated factory
```

3. Chain of Responsibility (Authentication)

```
public interface AuthenticationHandler {
    AuthenticationHandlerExecutionResult authenticate(Credential credential,
                                                      Service service) throws
                                                      ↪   Throwable;

    boolean supports(Credential credential);
    boolean supports(Class<? extends Credential> clazz);
}

// Handlers tried in chain until one succeeds
// - LdapAuthenticationHandler
// - DatabaseAuthenticationHandler
// - RadiusAuthenticationHandler
// - (50+ handlers)
```

4. Template Method (Webflow Actions)

```
// File:
//   core/cas-server-core-webflow-api/.../actions/BaseCasWebflowAction.java
```

```java
public abstract class BaseCasWebflowAction extends AbstractAction {

    @Override
    protected final Event doExecute(final RequestContext requestContext) throws
    ↪   Exception {
        // Publishes events, handles error wrapping, then delegates
        return doExecuteInternal(requestContext);
    }

    protected abstract Event doExecuteInternal(RequestContext requestContext)
    ↪   throws Throwable;
}
```

Subclasses implement `doExecuteInternal()` with specific logic while the base class handles event publishing and error handling.

5. Builder Pattern (Access Strategy Evaluation)

```
RegisteredServiceAccessStrategyEvaluator.builder()
    .registeredService(registeredService)
    .service(service)
    .authentication(authentication)
    .build()
    .execute();
```

6. Adapter Pattern (Protocol Engines)

Each protocol (CAS, SAML, OAuth, OIDC) adapts its specific requests to core ticket operations:

```
SAML Request -> Adapter -> createTicketGrantingTicket()
OAuth Request -> Adapter -> grantServiceTicket()
```

1.10 Component Interaction: A Typical Authentication Flow

Let’s trace how components interact during a basic CAS authentication:

1. User Accesses Protected Service

```
[Service] -> Redirect -> [CAS Login Page]
```

2. CAS Spring Webflow Activates

```
WebflowConfigurer -> Start "login" flow
  ├── InitialFlowSetupAction (prepare request context)
  ├── InitializeLoginAction (prepare credential)
  └── Render login view
```

3. User Submits Credentials

```
Form POST -> AuthenticationViaFormAction
  ├── Extract Credential from form
  ├── Invoke AuthenticationManager
  │    ├── AuthenticationHandlerResolver (determine applicable handlers)
  │    ├── AuthenticationEventExecutionPlan
  │    │    ├── LdapAuthenticationHandler.authenticate()
  │    │    └── [or other handlers...]
  │    ├── PrincipalResolver (build Principal from authenticated credential)
  │    └── AuthenticationPolicy (verify policies met)
  └── Build AuthenticationResult
```

4. Ticket Creation

```
AuthenticationResult -> CentralAuthenticationService.createTicketGrantingTicket()
  ├── TicketFactory.get(TicketGrantingTicket.class)
  ├── Create TGT with authentication
  ├── TicketRegistry.addTicket(tgt)
  └── Return TGT ID
```

5. Service Ticket Issuance

```
TGT + Service -> CentralAuthenticationService.grantServiceTicket()
  ├── Retrieve TGT from TicketRegistry
  ├── Validate TGT not expired
  ├── Check Service authorized (RegisteredServiceAccessStrategyAuditableEnforcer)
  ├── Create ServiceTicket via TicketFactory
  ├── TicketRegistry.addTicket(st)
  └── TicketRegistry.updateTicket(tgt) [track ST as child]
```

6. Redirect to Service

```
[CAS] -> HTTP 302 -> [Service]?ticket=ST-...
```

7. Service Validates Ticket

```
[Service] -> HTTP GET -> [CAS]/serviceValidate?ticket=ST-...&service=...
  ├── CentralAuthenticationService.validateServiceTicket()
  ├── TicketRegistry.getTicket(ST-...)
  ├── Validate ST matches service
  ├── TicketRegistry.deleteTicket(ST-...)
  ├── AttributeReleasePolicy.getAttributes()
  └── Return Assertion (Principal + Attributes)
```

Key Takeaways:

- Each step involves multiple Spring beans
- Interfaces allow swapping implementations
- Ticket registry is the central state store
- Authentication is pluggable and policy-driven
- Service validation is one-time use (ST deleted after validation)

1.11 Execution Flow Diagram

Here's how the major subsystems interact:

```
                          ┌──────────────────────┐
                          │   CasWebApplication  │
                          │   (Spring Boot)      │
                          └───────────┬──────────┘
                                      │
                    ┌─────────────────┼─────────────────┐
                    │                 │                 │
       ┌────────────▼────┐ ┌──────────▼───┐ ┌───────────▼───┐
       │ Auto-Config     │ │  Webflow     │ │  REST API     │
       │ Classes         │ │  Engine      │ │  Endpoints    │
```

```
  └──────────┬───────┘ └─────┬──────┘ └──────┬─────┘
             │               │               │
  ┌──────────▼───────────────▼───────────────▼─────┐
  │     CentralAuthenticationService               │
  │     (Core Orchestrator)                        │
  └──────────┬──────────────────────────┬──────────┘
             │                          │
┌────────────▼────────────┐  ┌──────────▼──────────┐
│ AuthenticationManager   │  │  TicketRegistry     │
│ (Handler Chains)        │  │  (State Storage)    │
└────────────┬────────────┘  └──────────┬──────────┘
             │                          │
┌────────────▼────────────┐  ┌──────────▼──────────┐
│ PrincipalResolver       │  │  Ticket Factories   │
│ (Attribute Resolution)  │  │  (TGT, ST, etc.)    │
└─────────────────────────┘  └─────────────────────┘
```

1.12 Performance Considerations

Module Count vs. Runtime Overhead

Question: With ~400 modules, doesn't this create massive overhead?

Answer: Not significantly, because:

1. **Only active modules load**: If you don't include Redis support, those classes never load
2. **Conditional beans**: `@ConditionalOnProperty` prevents unnecessary bean creation
3. **Lazy initialization**: Many beans are lazy-loaded on first use
4. **proxyBeanMethods = false**: Reduces CGLIB overhead
5. **GraalVM native image**: CAS can compile to native binary for instant startup

Startup Time Optimization

CAS 7.x improves startup time via:

- **Virtual threads** (Java 21+): Replace thread pools with cheaper virtual threads
- **Class data sharing**: JVM CDS reduces class loading time
- **Spring Boot 3.x**: Improved auto-configuration processing
- **Lazy bean initialization**: Defer bean creation until needed

Memory Footprint

- **Default configuration**: ~500MB heap for basic deployment
- **With JPA + Redis + SAML**: ~1GB heap
- **Production recommendation**: 2-4GB heap for headroom

1.13 Extension Points

CAS is designed to be extended. Here are the primary extension mechanisms:

1. Custom Beans

Override any bean by defining your own with the same name:

```
@Configuration
public class MyCustomConfiguration {
    @Bean
    public AuthenticationHandler myCustomAuthHandler() {
        return new MyAuthHandler();
    }
}
```

2. Auto-Configuration Contributions

Add your own auto-configuration class:

```
META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports:
com.example.MyCustomAutoConfiguration
```

3. Implementing Core Interfaces

Implement key interfaces:

- `AuthenticationHandler` (custom authentication)
- `TicketRegistry` (custom storage)
- `PrincipalResolver` (custom attribute resolution)
- `RegisteredServiceAccessStrategy` (custom authorization)

4. Webflow Customization

Add states, transitions, actions to login flow:

```
@Component
public class MyWebflowConfigurer extends AbstractCasWebflowConfigurer {
    @Override
    protected void doInitialize() {
        val flow = getLoginFlow();
        createTransitionForState(flow,
        ↪   "initialAuthenticationRequestValidationCheck",
                                      "myCustomState", "customCondition");
    }
}
```

5. Event Listeners

React to CAS events:

```
@Component
public class MyEventListener {
    @EventListener
    public void handleAuthSuccess(CasAuthenticationTransactionSuccessfulEvent
    ↪   event) {
        // Custom logic
    }
}
```

1.14 Testing Strategies

Module-Level Testing

Each module contains:

1. **Unit tests**: Test individual classes in isolation
2. **Integration tests**: Test Spring context loading
3. **Test configurations**: Reusable test beans

Example:

```
@SpringBootTest(classes = {
    CasCoreAuthenticationAutoConfiguration.class,
    // Include only configurations needed for test
})
class AuthenticationManagerTests {
    @Autowired
    private AuthenticationManager authenticationManager;

    @Test
    void testAuthentication() {
        // ...
    }
}
```

Conditional Testing

CAS uses `@ConditionalOnProperty` in tests:

```
@ConditionalOnProperty(name = "cas.authn.ldap.enabled", havingValue = "true")
@SpringBootTest
class LdapAuthenticationTests {
    // Only runs when LDAP is configured
}
```

Test Utilities

CAS provides extensive test utilities:

- `RegisteredServiceTestUtils`: Create test service definitions

- `CoreAuthenticationTestUtils`: Create test authentication objects
- `MockTicketGrantingTicket`: Mock tickets
- Test property files with embedded servers (LDAP, Redis, databases)

1.15 Common Pitfalls

Pitfall 1: Missing Dependencies

Problem: Auto-configuration fails because optional dependency not included

Example: Using Redis ticket registry without `spring-boot-starter-data-redis`

Why: CAS modules declare dependencies as `compileOnly` to keep WAR size manageable

Solution: Explicitly include dependencies in your `build.gradle`/`pom.xml`

Pitfall 2: Bean Name Conflicts

Problem: Custom bean doesn't override CAS bean

Why: Bean name mismatch (CAS uses explicit `@Bean(name = "...")`)

Solution: Use exact bean name from CAS configuration class:

```
@Bean(name = "ticketRegistry")  // Exact name CAS expects
public TicketRegistry myCustomRegistry() {
    return new MyTicketRegistry();
}
```

Pitfall 3: Auto-Configuration Ordering

Problem: Custom configuration runs before/after CAS configuration

Why: Spring Boot auto-configuration order is non-deterministic by default

Solution: Use `@AutoConfigureBefore` / `@AutoConfigureAfter`:

```
@AutoConfiguration
@AutoConfigureAfter(CasCoreTicketsAutoConfiguration.class)
public class MyTicketConfiguration {
    // Runs after CAS ticket configuration
}
```

Pitfall 4: Property Precedence Confusion

Problem: Properties not taking effect

Why: CAS uses Spring Boot property precedence: 1. Command-line args 2. System properties 3. `application.properties`/`application.yml` 4. `cas.standalone.configurationDirectory` files 5. Embedded defaults

Solution: Check all property sources with actuator `/env` endpoint

---

1.16 Debugging Tips

Viewing Auto-Configuration Report

Enable debug logging to see what auto-configures:

```
java -jar cas.war --debug
```

Look for:

```
============================
CONDITIONS EVALUATION REPORT
============================

Positive matches:
-----------------
   CasCoreAuthenticationAutoConfiguration matched:
      - @ConditionalOnMissingBean (...)

Negative matches:
-----------------
```

```
  RedisTicketRegistryAutoConfiguration:
     Did not match:
        - @ConditionalOnClass did not find required class
        'org.springframework.data.redis.core.RedisTemplate'
```

Useful Logging Categories

```
logging.level.org.apereo.cas=DEBUG                      # All CAS
logging.level.org.apereo.cas.authentication=TRACE      # Authentication flow
logging.level.org.apereo.cas.ticket=DEBUG               # Ticket operations
logging.level.org.apereo.cas.services=DEBUG             # Service registry
logging.level.org.springframework.webflow=DEBUG         # Webflow execution
logging.level.org.springframework.boot.autoconfigure=DEBUG  # Auto-config
decisions
```

Actuator Endpoints for Introspection

Enable Spring Boot Actuator:

```
management.endpoints.web.exposure.include=*
```

Then access:

- `/actuator/beans`: See all registered beans
- `/actuator/conditions`: Auto-configuration report
- `/actuator/configprops`: All bound configuration properties
- `/actuator/env`: All property sources
- `/actuator/mappings`: All HTTP endpoints

Breakpoint Strategies

Key places to set breakpoints:

1. **Bootstrap**: `CasWebApplication.main()`
2. **Auto-config**: Any `@AutoConfiguration` class's `@Bean` methods
3. **Authentication**: `AuthenticationManager.authenticate()`
4. **Ticket creation**: `CentralAuthenticationService.createTicketGrantingTicket()`

5. **Webflow transitions**: `Action.execute()` implementations

1.17 Related Components

Understanding CAS architecture sets the foundation for all subsequent chapters:

- **Chapter 2: Spring Boot Bootstrap & Auto-Configuration** - Spring Boot bootstrap and configuration binding
- **Chapter 3: Configuration Management Implementation** - How configuration properties drive component assembly
- **Chapter 4: Authentication Flow** - Authentication flow through the component stack
- **Chapter 5: Spring Webflow State Machine** - Webflow state machine architecture
- **Chapter 7: Ticket Registry Architecture** - Ticket registry implementations
- **Chapter 10: Service Registry System** - Service registry architecture
- **Chapter 29: Extension Points & Customization** - Creating custom extensions using these patterns

---

1.18 Summary

- **CAS is modular**: ~400 modules organized as API -> Core -> Support layers
- **Spring Boot drives assembly**: Auto-configuration wires components based on classpath and properties
- **Interface-driven design**: Core abstractions (`CentralAuthenticationService`, `TicketRegistry`, `AuthenticationHandler`) allow pluggability
- **Convention over configuration**: Sensible defaults with override capabilities via bean replacement

- **Three-layer dependency flow**: API <- Core <- Support prevents circular dependencies
- **Extensibility by design**: Override beans, contribute auto-configurations, implement interfaces
- **Testing is first-class**: Each module is independently testable with provided test utilities

With this architectural foundation, you're ready to explore specific subsystems knowing how they fit into the larger picture.

1.19 Further Reading

CAS Documentation:

- Official architecture overview: https://apereo.github.io/cas/
- Module structure guide: https://apereo.github.io/cas/development/

Source Code References:

- `api/` - Interface definitions and contracts
- `core/` - Default implementations (45 modules)
- `support/` - Optional integrations (348 modules)
- `webapp/cas-server-webapp-init/` - Bootstrap entry point

Related Chapters:

- Chapter 2: Spring Boot Bootstrap & Auto-Configuration
- Chapter 3: Configuration Management Implementation
- Chapter 29: Extension Points & Customization
- Chapter 30: Module Development
